Python Programming Exercises, Gently Explained

Al Sweigart

Author of *Automate the Boring Stuff with Python*

InventWithPython.com

Foreword by Trey Hunner

Python Programming Exercises, Gently Explained.

Copyright © 2022 by Al Sweigart. All rights reserved.

No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, beyond the Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International license without the prior written permission of the copyright owner.

ISBN-13: 979-8-3553-8768-6

Cover Illustration: Al Sweigart

For information about this book and its content, please contact al@inventwithpython.com.

Second printing.

The information in this book is distributed on an "As Is" basis, without warranty. While every precaution has been taken in the preparation of this work, the author shall not have any liability for any loss or damage caused or alleged to be caused directly or indirectly by the information contained in it.

This work is licensed under the Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International (CC BY-NC-SA 4.0) license. For a copy of this license, visit https://creativecommons.org/licenses/by-nc-sa/4.0/ or send a letter to Creative Commons, PO Box 1866, Mountain View, CA 94042, USA.

DEDICATION

To Laura and Alexandra, and the rest of the next generation.

TABLE OF CONTENTS

FOREWORD

I met Al Sweigart on a chilly April day in Montreal, Canada. It was PyCon 2015 and I got together for lunch with a few new conference friends—including Al. I immediately clicked with him because he's a teacher, he enjoys playing with ideas, he's concerned about the world, and he has an absurd (read: wonderful) sense of humor. When I met Al, I didn't know he was famous, and neither did he. It was during another meal when a reader of *Automate the Boring Stuff with Python* who had been sitting with us discovered that the Al at our table was the Al Sweigart.

New conference friend: "Wait, you're Al Sweigart? Your book is amazing!"

Me: "Al, you didn't tell me you were famous."

Al: "I didn't know I was famous either!"

If this book is your first introduction to Al, you should know that he's a pragmatic Python programmer who is marvelous at embracing Python's power to churn out code that solves real world problems. I don't always agree with Al's code style choices (camelCase in Python, Al?!) but Al comments his code clearly and breaks down problems succinctly, and I know many Python programmers who were introduced to the beauty of Python through Al's work.

So now you know that I'm in the Al Sweigart fan club. But who am I, and why am I writing the foreword for this book?

My name is Trey Hunner. I'm a Python team trainer and I run Python Morsels, which helps Python developers grow their skills through Python exercises and detailed solution walk-throughs. I started a Python exercise service because I know that the most important part of my team training sessions is exercise time.

In fact, there's a mantra I repeat to my students before every Python exercise session: We don't learn by putting information into our heads; we learn by trying to retrieve information from our heads.

Learning happens when we attempt to use our memory. Whether we're remembering rote facts, practicing muscle memory, or attempting to identify which tool in our mental toolbox applies to the situation at hand. You can watch YouTube videos on Python all you want, but you'll quickly forget each new Python feature you see unless you actually use those features in your own code.

If you want to learn Python, you need to write Python code. That's where Python exercises (and this book) come in. You can't grow your Python skills by writing yet another "hello world" program. You need to write code that pushes you *just* outside your learning comfort zone ("the zone of proximal development", as learning nerds call it).

I hope you'll find a few exercises that push you out of your comfort zone in this short book—written by *the* Al Sweigart.

Trey Hunner, *https://treyhunner.com/*

INTRODUCTION

"How can I get better at programming?" I see this common question often from those who have started their programming journey. Of course, there are plenty of Python programming tutorials for total beginners. However, these tutorials can carry the reader so far. After finishing these lessons, readers often find their skills more than capable for yet another "Hello, world!" tutorial but not advanced enough to begin writing their own programs. They find themselves in the so-called "tutorial hell." They learn the basic syntax of a programming language, but wonder where to begin when it comes to applying them to their own programs.

Programming is like any other skill: it gets better with practice. I've chosen the exercises in this book because they are short and straightforward. Each exercise involves only a handful of programming concepts and you can solve them in a single session at the computer. If you've been intimidated by "competitive programming" or "hacker challenge" websites, you'll find these to be an instructive and gentler way to level up your coding skills.

When I wrote *Automate the Boring Stuff with Python*, I wanted to teach programming to as many non-programmers as possible. So, I released that book (and all of my other programming books) under a Creative Commons license so they could be downloaded and shared for free from my website *https://inventwithpython.com*. You can pay for a physical print book but it's important to me to lower all barriers to access, so the books are available online for free. I'll cite some of these books in the **Further Reading** section of the exercises in this book.

What Will This Book Do For You?

This book offers 42 programming exercises for inexperienced Python programmers. I've gathered them into this book and combined them with plain-English explanations. You can read the description for each exercise and start on the solution immediately. If you need further help, you can read about the programming concepts you'll need to know for the solution. You can also find out about any surprising "gotchas" you might encounter while writing your solution. Finally, if you still need help, I provide a fill-in-the-blank template of the solution. Try to resist the temptation to immediately jump to the hints; try to solve these exercises yourself first.

As you work through these exercises, you'll find that some use the same coding techniques as other problems. A lot of programming expertise develops this way: being able to solve a problem isn't about how smart you are but if you've seen similar problems before. My aim isn't to stump you with complex, contrived programming challenges but help you explore simple problems with gentle explanations.

Prerequisites

While this isn't a book to teach programming to complete beginners, you don't need to be a programming expert before tackling the challenges here. Any beginner's resource, such as one of my free books, *Automate the Boring Stuff with Python* or *Invent Your Own Computer Games with Python*, is more

than enough to prepare you for these exercises. You can find these books for free at *https://inventwithpython.com*. I also recommend *Python Crash Course* by Eric Matthes as an excellent book for people with no programming experience. These books are published by No Starch Press, which has an extensive library of high-quality Python books at *https://nostarch.com/catalog/python*.

To solve the exercises in this book, you should already know, or at least be familiar with, the following Python programming concepts:

- Downloading and installing the Python interpreter
- Entering Python code into the interactive shell and into *.py* source code files
- Storing values in variables with the `=` assignment operator
- Knowing the difference between data types such as integers, strings, and floats
- Math, comparison, and Boolean operators such as `+`, `<=`, and `not`.
- How Python evaluates expressions such as `(2 * 3) + 4` or `'Sun' + 'day'`
- Getting and displaying text with the `input()` and `print()` functions
- Working with strings and string methods such as `upper()` or `startswith()`
- Using `if`, `elif`, and `else` flow control statements
- Using `for` loops and `while` loops, along with `break` and `continue` statements
- Defining your own functions with parameters and returning values
- Using data structures such as lists, tuples, and dictionaries
- Importing modules such as `math` or `random` to use their functions

You don't need a mastery of classes and object-oriented programming. You don't need to know advanced concepts such as machine learning or data science. But if you'd like to eventually build your skills to advanced topics like these, this book can help you start along that path.

Even if you've moved past the beginner stage, this book can help you assess your programming ability. By the time you're ready to start applying to junior software developer positions, you should be able to solve all of the exercises in this book easily.

About the Exercises

The exercises are generally ordered from least to most difficult. But you don't have to solve them in order, so feel free to jump around to any exercises that you find interesting.

Each exercise has the following sections:

- **Exercise Description** – A description of the exercise, followed by a list of `assert` statements that specify the results it expects from your solution program. There is also a list of prerequisite concepts you'll need to understand to solve this exercise. If you don't understand any of them, you can do an internet search of these terms along with "Python" to find explanations of them. This is the only section you need to read to solve the exercise. The later sections provide additional hints if you need them.
- **Solution Design** – Additional information about concepts you'll need to know to write a solution, along with a brief, gentle explanation of them.

- **Special Cases and Gotchas** – Describes common mistakes or surprising "gotchas" you may encounter when writing code for the solution. Some exercises have special cases that your solution will need to address.
- **Solution Template** – A copy of my own solution for the exercise, with selected parts replaced by blanks for you to fill in. Your solution can still be correct if it doesn't match mine. But if you're having trouble knowing where to start with your program, these templates provide some but not all of the solution code.

Many solution programs to the exercises in this book are only a few lines of code long, and none of them are longer than 50 lines. If you find yourself writing several hundred lines of code, you're probably overthinking the solution and should probably read the **Exercise Description** section again.

Each exercise has several `assert` statements that detail the expected results from your solution. In Python, `assert` statements are the `assert` keyword followed by a condition. They stop the program with an `AssertionError` if their condition is `False`. They are a basic sanity check and, for this book, tell you the expected behavior of your solution. For example, Exercise #3, "Odd & Even" has assert `isOdd(9999) == True`, which tells you that the correct solution involves the `isOdd()` function returning `True` when passed an argument of `9999`. Examine all of the `assert` statements for an exercise before writing your solution program.

Some exercises don't have `assert` statements, but rather show you the output that your solution should produce. You can compare this output to your solution's output to verify that your solution is correct.

I provide complete solutions in Appendix A. But there are many ways to solve any given programming problem. You don't need to produce an identical copy of my solutions; it just needs to pass the `assert` statements. I wrote my solutions to be conceptually simple and easy for the intended audience of this book to understand. They produce correct results but aren't necessarily the fastest or most efficient solutions. As you get more experience programming, you can revisit the exercises in this book and attempt to write high-performance solutions to them.

Most of the solutions involve writing functions that return values based on the arguments passed to the function call. In these cases, you can write your code assuming that the arguments are always of the expected data type. So for example, if your function expects an integer, it will have to handle arguments like `42`, `0`, or `-3` but doesn't have to handle arguments like `3.14` or `'hello'`.

Keep in mind that there is a difference between a *parameter* and an *argument*. When we *define* a function such as Exercise #3's `def isOdd(number):`, the local variable `number` is a *parameter*. When we call this function with `isOdd(42)`, the integer `42` is an *argument*. The argument is passed to the function and assigned as the value to the parameter. It's easy to use "parameter" and "argument" interchangeably, but this book uses them in their correct technical sense.

Python is a practical language with many helpful functions in its standard library. Some exercises will explicitly forbid you from using these functions. For example, Exercise #34, "Uppercase Letters" tasks you to write code to convert a string to uppercase letters. Using Python's built-in `upper()` string method to do this for you would defeat the purpose of the exercise, so the **Exercise Description** section points out that your solution shouldn't call it.

I recommend solving the exercises in this book repeatedly until it becomes effortless. If you can solve an exercise once, try solving it again a couple of weeks later or without looking at the hints in the later sections. After a while, you'll find yourself quickly being able to come up with strategies to solve these exercises.

Let's begin!

EXERCISE #1: HELLO, WORLD!

```
print('Hello, world!')  →  Hello, world!
```

"Hello, world!" is a common first program to write when learning any programming language. It makes the text "Hello, world!" appear on the screen. While not much of a program, it serves as a simple test for whether the programmer has the language interpreter correctly installed, the computer environment set up, and a basic understanding of how to write a complete program.

This exercise adds an additional step to collect keyboard input from the user. You'll also need to concatenate (that is, join together) string values to greet the user by name. The exercises in this book are for those with some experience writing programs, so if this exercise is currently beyond your skill level, I'd recommend reading some of the beginner resources I discussed in the **Prerequisites** section of the Introduction.

Exercise Description

Write a program that, when run, greets the user by printing "Hello, world!" on the screen. Then it prints a message on the screen asking the user to enter their name. The program greets the user by name by printing the "Hello," followed by the user's name.

Let's use "Alice" as the example name. Your program's output should look like this:

```
Hello, world!
What is your name?
Alice
Hello, Alice
```

Try to write a solution based on the information in this description. If you still have trouble solving this exercise, read the **Solution Design** and **Special Cases and Gotchas** sections for additional hints.

Prerequisite concepts: `print()`, strings, string concatenation

Solution Design

There is no surprising design to the solution. It is a straightforward four-line program:

1. Print the text, "Hello, world!"
2. Print the text, "What is your name?"
3. Let the user enter their name and store this in a variable.
4. Print the text "Hello," followed by their name.

The program calls Python's `print()` function to display string values on the screen. As always, the quotes of the string aren't displayed on the screen because those aren't part of the string value's text, but rather mark where the string begins and ends in the source code of the program. The program calls Python's `input()` function to get the user input from the keyboard and stores this in a variable.

The `+` operator is used to add numeric values like integers or floating-point numbers together, but it can also create a new string value from the concatenation of two other string values.

Special Cases and Gotchas

Make sure that there is a space before printing the user's name. If the user entered "Alice", the program should print `'Hello, Alice'` and not `'Hello,Alice'`.

Now try to write a solution based on the information in the previous sections. If you still have trouble solving this exercise, read the **Solution Template** section for additional hints.

Solution Template

Try to first write a solution from scratch, But if you have difficulty, you can use the following partial program as a starting place. Copy the following code from *https://invpy.com/helloworld-template.py* and paste it into your code editor. Replace the underscores with code to make a working program:

```
# Print "Hello, world!" on the screen:
____('Hello, world!')
# Ask the user for their name:
____('What is your name?')
# Get the user's name from their keyboard input:
name = ____()
# Greet the user by their name:
print('Hello, ' + ____)
```

The complete solution for this exercise is given in Appendix A and *https://invpy.com/helloworld.py*. You can view each step of this program as it runs under a debugger at *https://invpy.com/helloworld-debug/*.

Further Reading

A "Hello, world!" program is a good target program to create when learning a new programming language. Most languages have a concept of standard streams for text-based, command-line programs. These programs have a stream of input text and a stream of output text. The stream of output text the program produces appears on the screen (via `print()` in Python). The stream of

input text the program accepts comes from the keyboard (via `input()` in Python). The main benefit of streams is that you can redirect them: the text output can go to a file instead of the screen, or a program's input can come from the output of another program instead of the keyboard. You can learn more about the *command-line interface*, also called *terminal windows*, in the free book "Beyond the Basic Stuff with Python" at *https://inventwithpython.com/beyond/*. Command-line programs tend to be simpler than graphical programs, and you can find nearly a hundred such programs in the free book, *The Big Book of Small Python Projects* at *https://inventwithpython.com/bigbookpython/*.

EXERCISE #2: TEMPERATURE CONVERSION

```
convertToCelsius(32)  →  0.0
convertToFahrenheit(100)  →  212
```

Converting between Celsius and Fahrenheit involves a basic calculation and provides a good exercise for writing functions that take in a numeric input and return a numeric output. This exercise tests your ability to use Python's math operators and translate math equations into Python code.

Exercise Description

Write a `convertToFahrenheit()` function with a `degreesCelsius` parameter. This function returns the number of this temperature in degrees Fahrenheit. Then write a function named `convertToCelsius()` with a `degreesFahrenheit` parameter and returns a number of this temperature in degrees Celsius.

Use these two formulas for converting between Celsius and Fahrenheit:

- Fahrenheit = Celsius × (9 / 5) + 32
- Celsius = (Fahrenheit - 32) × (5 / 9)

These Python `assert` statements stop the program if their condition is `False`. Copy them to the bottom of your solution program. Your solution is correct if the following `assert` statements' conditions are all `True`:

```
assert convertToCelsius(0) == -17.77777777777778
assert convertToCelsius(180) == 82.22222222222223
assert convertToFahrenheit(0) == 32
assert convertToFahrenheit(100) == 212
assert convertToCelsius(convertToFahrenheit(15)) == 15

# Rounding errors cause a slight discrepancy:
assert convertToCelsius(convertToFahrenheit(42)) == 42.00000000000001
```

Try to write a solution based on the information in this description. If you still have trouble solving this exercise, read the **Solution Design** and **Special Cases and Gotchas** sections for additional hints.

Prerequisite concepts: math operators

Solution Design

Think about how you can translate the mathematical equations in the Exercise Description into Python code. For converting from Celsius to Fahrenheit, the "Celsius" is the `degreesCelsius` parameter, the × multiplication sign can be replaced by Python's `*` operator, and the parentheses and other math symbols have direct Python equivalents. You can think of the "Fahrenheit =" and "Celsius =" parts in the formula to Python `return` statements.

These functions can be one line long: a `return` statement that calculates the formula's result and returns it.

Special Cases and Gotchas

When converting 42 degrees Celsius to Fahrenheit and back to Celsius in the Assertions section, you may have noticed that you don't get exactly 42 but 42.00000000000001. This tiny fractional part is caused by rounding errors. Computers cannot perfectly represent all numbers with fractional parts. For example, if you enter `print(0.1 + 0.1 + 0.1)` into Python's interactive shell, it doesn't print `0.3` but rather prints `0.30000000000000004`.

The Python programming language doesn't cause these rounding errors. Instead, they come from the technical standard for floating-point numbers used by all computers. Every programming language has these same rounding errors. However, unless you are writing software for a bank or nuclear reactor, these minor inaccuracies probably don't matter. The **Further Reading** section

Now try to write a solution based on the information in the previous sections. If you still have trouble solving this exercise, read the **Solution Template** section for additional hints.

Solution Template

Try to first write a solution from scratch. But if you have difficulty, you can use the following partial program as a starting place. Copy the following code from *https://inventwithpython.com/converttemp-template.py* and paste it into your code editor. Replace the underscores with code to make a working program:

```
def convertToFahrenheit(degreesCelsius):
    # Calculate and return the degrees Fahrenheit:
    return ____ * (9 / 5) + 32


def convertToCelsius(degreesFahrenheit):
    # Calculate and return the degrees Celsius:
    return (____ - 32) * (____ / ____)
```

The complete solution for this exercise is given in Appendix A and *https://inventwithpython.com/converttemp.py*. You can view each step of this program as it runs under a debugger at *https://inventwithpython.com/converttemp-debug/*.

Further Reading

You can use Python's `decimal` module to represent fractional numbers more accurately than floating-point values can. You can learn about this module at the Python Module of the Week blog at *https://pymotw.com/3/decimal/*.

One programming technique to avoid rounding errors is to stick with integers: instead of representing $1.25 as the float `1.25` you can use the integer `125` for 125 cents, or instead of representing 1.5 minutes as the float `1.5` you can use the integer `90` for 90 seconds or even `90000` for 90,000 milliseconds. Integers and the `decimal` module's `Decimal` objects don't suffer from rounding errors like floating-point numbers do.

EXERCISE #3: ODD & EVEN

```
isOdd(13)  →  True
isEven(13)  →  False
```

Determining if a number is even or odd is a common calculation that uses the modulo operator. Like Exercise #2, "Temperature Conversion," the functions for this exercise's solution functions can be as little as one line long.

This exercise covers the % modulo operator, and the technique of using modulo-2 arithmetic to determine if a number is even or odd.

Exercise Description

Write two functions, `isOdd()` and `isEven()`, with a single numeric parameter named `number`. The `isOdd()` function returns `True` if `number` is odd and `False` if `number` is even. The `isEven()` function returns the `True` if `number` is even and `False` if `number` is odd. Both functions return `False` for numbers with fractional parts, such as `3.14` or `-4.5`. Zero is considered an even number.

These Python `assert` statements stop the program if their condition is `False`. Copy them to the bottom of your solution program. Your solution is correct if the following `assert` statements' conditions are all `True`:

```
assert isOdd(42) == False
assert isOdd(9999) == True
assert isOdd(-10) == False
assert isOdd(-11) == True
assert isOdd(3.1415) == False
assert isEven(42) == True
assert isEven(9999) == False
assert isEven(-10) == True
assert isEven(-11) == False
assert isEven(3.1415) == False
```

Try to write a solution based on the information in this description. If you still have trouble solving this exercise, read the **Solution Design** and **Special Cases and Gotchas** sections for additional hints.

Prerequisite concepts: modulo operator

Solution Design

You might not know how to write code that finds out if a number is odd or even. This kind of exercise is easy to solve if you already know how to solve it, but difficult if you have to reinvent the solution yourself. Feel free to look up answers to questions you have on the internet. Include the programming language's name for more specific results, such as "*python find out if a number is odd or even*". The question-and-answer website *https://stackoverflow.com* is usually at the top of the search results and reliably has direct, high-quality answers. Looking up programming answers online isn't "cheating." Professional software developers look up answers dozens of times a day!

The `%` modulo operator can mod a number by `2` to determine its evenness or oddness. The modulo operator acts as a sort of "division remainder" operator. If you divide a number by `2` and the remainder is `1`, the number must be odd. If the remainder is `0`, the number must be even. For example, `42 % 2` is `0`, meaning that `42` is even. But `7 % 2` is `1`, meaning that `7` is odd.

Floating-point numbers such as `3.1415` are neither odd nor even, so both `isOdd()` and `isEven()` should return `False` for them.

Keep in mind that the `isOdd()` and `isEven()` function you write must return a Boolean `True` or `False` value, not an integer `0` or `1`. You need to use a `==` or `!=` comparison operator to produce a Boolean value: `7 % 2 == 1` evaluates to `1 == 1`, which in turn evaluates to `True`.

Special Cases and Gotchas

Non-integer numbers such as `4.5` or `3.1415` are neither odd nor even, so both `isOdd()` and `isEven()` should return `False` for them.

Now try to write a solution based on the information in the previous sections. If you still have trouble solving this exercise, read the **Solution Template** section for additional hints.

Solution Template

Try to first write a solution from scratch. But if you have difficulty, you can use the following partial program as a starting place. Copy the following code from *https://invpy.com/oddeven-template.py* and paste it into your code editor. Replace the underscores with code to make a working program:

```
def isOdd(number):
    # Return whether number mod 2 is 1:
    return ____ % 2 == ____

def isEven(number):
    # Return whether number mod 2 is 0:
    return ____ % 2 == ____
```

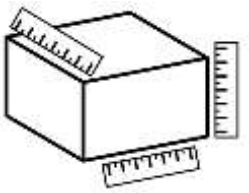
The complete solution for this exercise is given in Appendix A and *https://invpy.com/oddeven.py*. You can view each step of this program as it runs under a debugger at *https://invpy.com/oddeven-debug/*.

Further Reading

There are several uses of the `%` modulo operator. You can learn more about them in the tutorial "Python Modulo in Practice: How to Use the % Operator" at *https://realpython.com/python-modulo-operator/*.

EXERCISE #4: AREA & VOLUME

```
      area(10, 4)  →  40
perimeter(10, 4)  →  28
   volume(10, 4, 5)  →  200
surfaceArea(10, 4, 5)  →  220
```



Area, perimeter, volume, and surface area are straightforward calculations. This exercise is similar to Exercise #2, "Temperature Conversion" and Exercise #3, "Odd & Even." Each function in this exercise is a simple calculation and `return` statement. However, area and volume are slightly more complicated because they involve multiple parameters. This exercise continues to challenge you to translate mathematical formulas into Python code.

Exercise Description

You will write four functions for this exercise. The functions `area()` and `perimeter()` have `length` and `width` parameters and the functions `volume()` and `surfaceArea()` have `length`, `width`, and `height` parameters. These functions return the area, perimeter, volume, and surface area, respectively.

The formulas for calculating area, perimeter, volume, and surface area are based on the length (L), width (W), and height (H) of the shape:

- area = L × W
- perimeter = L + W + L + W
- volume = L × W × H
- surface area = (L × W × 2) + (L × H × 2) + (W × H × 2)

Area is a two-dimensional measurement from multiplying length and width. Perimeter is the sum of all of the four one-dimensional lines around the rectangle. Volume is a three-dimensional measurement from multiplying length, width, and height. Surface area is the sum of all six of the two-dimensional areas around the cube. Each of these areas comes from multiplying two of the three length, width, or height dimensions.

You can see what these formulas measure in Figure 4-1.

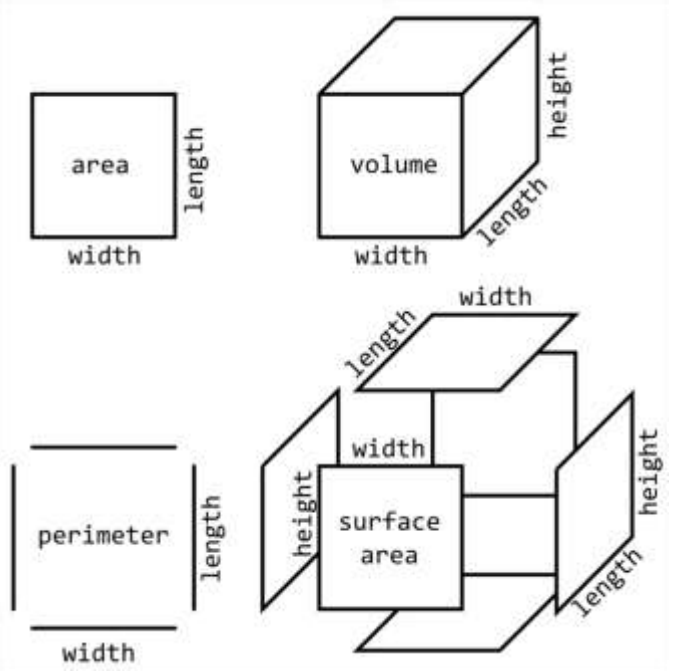


Figure 4-1: The components of area, volume, perimeter, and surface area.

These Python `assert` statements stop the program if their condition is `False`. Copy them to the bottom of your solution program. Your solution is correct if the following `assert` statements' conditions are all `True`:

```
assert area(10, 10) == 100
assert area(0, 9999) == 0
assert area(5, 8) == 40
assert perimeter(10, 10) == 40
assert perimeter(0, 9999) == 19998
assert perimeter(5, 8) == 26
assert volume(10, 10, 10) == 1000
assert volume(9999, 0, 9999) == 0
assert volume(5, 8, 10) == 400
assert surfaceArea(10, 10, 10) == 600
assert surfaceArea(9999, 0, 9999) == 199960002
assert surfaceArea(5, 8, 10) == 340
```

Try to write a solution based on the information in this description. If you still have trouble solving this exercise, read the **Solution Design** and **Special Cases and Gotchas** sections for additional hints.

Prerequisite concepts: math operators, operator precedence

Solution Design

Use the same strategy you used in Exercise #2, "Temperature Conversion" and translate the math formulas into Python code. Replace the L, W, and H with the `length`, `width`, and `height` parameters. Replace the + addition and × multiplication signs with the `+` and `*` Python operators. These functions can be one line long: a `return` statement that calculates the result and returns it.

Special Cases and Gotchas

Length, width, and height can only be positive numbers. You could write additional code that checks if any of these are negative numbers, and raise an exception in that case. However, this isn't a requirement for this exercise.

Now try to write a solution based on the information in the previous sections. If you still have trouble solving this exercise, read the **Solution Template** section for additional hints.

Solution Template

Try to first write a solution from scratch. But if you have difficulty, you can use the following partial program as a starting place. Copy the following code from *https://invpy.com/areavolume-template.py* and paste it into your code editor. Replace the underscores with code to make a working program:

```
def area(length, width):
    # Return the product of the length and width:
    return ____ * ____

def perimeter(length, width):
    # Return the sum of the length twice and the width twice:
    return ____ * 2 + width * ____

def volume(length, width, height):
    # Return the product of the length, width, and height:
    return ____ * ____ * ____

def surfaceArea(length, width, height):
    # Return the sum of the area of each of the six sides:
    return ((length * ____) + (length * ____) + (width * ____)) * 2
```

The complete solution for this exercise is given in Appendix A and *https://invpy.com/areavolume.py*. You can view each step of this program as it runs under a debugger at *https://invpy.com/areavolume-debug/*.

EXERCISE #5: FIZZ BUZZ

```
fizzBuzz(5)  →  1 2 Fizz 4 Buzz
```

Fizz buzz is a word game you can implement as a simple program. It became famous as a screening question in coding interviews to quickly determine if candidates had any programming ability whatsoever, so being able to solve it quickly leads to a good first impression.

This exercise continues our use of the modulo operator to determine if numbers are divisible by 3, 5, or both 3 and 5. "Divisible by *n*" means that it can be divided by a number *n* with no remainder. For example, 10 is divisible by 5, but 11 is not divisible by 5.

Exercise Description

Write a `fizzBuzz()` function with a single integer parameter named `upTo`. For the numbers `1` up to and including `upTo`, the function prints one of four things:

- Prints `'FizzBuzz'` if the number is divisible by 3 *and* 5.
- Prints `'Fizz'` if the number is only divisible by 3.
- Prints `'Buzz'` if the number is only divisible by 5.
- Prints the number if the number is *neither* divisible by 3 nor 5.

Instead of printing each string or number on a separate line, print them without newlines. For example, your solution is correct if calling `fizzBuzz(35)` produces the following output:

```
1 2 Fizz 4 Buzz Fizz 7 8 Fizz Buzz 11 Fizz 13 14 FizzBuzz 16 17 Fizz 19 Buzz Fizz 22
23 Fizz Buzz 26 Fizz 28 29 FizzBuzz 31 32 Fizz 34 Buzz
```

Try to write a solution based on the information in this description. If you still have trouble solving this exercise, read the **Solution Design** and **Special Cases and Gotchas** sections for additional hints.

Prerequisite concepts: modulo operator, `end` keyword argument for `print()`, for loops, `range()` with two arguments

Solution Design

This function requires a loop covering the integers `1` up to and including the `upTo` parameter. The `%` modulo operator (which we used in Exercise #3, "Odd & Even") can check if a number is divisible by 3 or 5. If a number mod 3 is 0, then the number is divisible by 3. If a number mod 5 is 0, then the number is divisible by 5. For example, enter the following into the interactive shell:

```
>>> 9 % 3
0
>>> 10 % 3
1
>>> 11 % 3
2
```

`9 % 3` evaluates to `0`, so 9 is divisible by 3. Meanwhile, `10 % 3` evaluates to `1` and `11 % 3` evaluates `2`, so 10 and 11 aren't divisible by 3. To see the pattern behind the modulo operator, enter the following code into the interactive shell:

```
>>> for i in range(20):
...   print(str(i).rjust(2), i % 3, i % 5)
...
 0 0 0
 1 1 1
 2 2 2
 3 0 3
 4 1 4
 5 2 0
 6 0 1
 7 1 2
 8 2 3
 9 0 4
10 1 0
11 2 1
12 0 2
13 1 3
14 2 4
15 0 0
16 1 1
17 2 2
18 0 3
19 1 4
```

The columns of three numbers are a number from 0 to 19, the number modulo 3, and the number modulo 5. Notice that the modulo 3 column cycles from 0 to 2 and the modulo 5 column cycles from 0 to 4. The modulo result is 0 every 3rd and 5th number, respectively. And they are both 0 every 15th number (more on this in the **Special Cases and Gotchas** section.)

Back to our solution, the code inside the loop checks if the current number should cause the program to either display one of "Fizz", "Buzz", "FizzBuzz", or the number. This can be handled with a series of `if`-`elif`-`elif`-`else` statements.

To tell the `print()` function to automatically print a space instead of a newline character, pass the `end=' '` keyword argument. For example, `print('FizzBuzz', end=' ')`

Special Cases and Gotchas

One common mistake when writing a Fizz Buzz program is checking if the number is divisible by

3 *or* 5 before checking if it's divisible by 3 *and* 5. You'll want to check if a number is divisible by 3 and 5 first, because these numbers are also divisible by `3` and `5`. But you want to be sure to display `'FizzBuzz'` rather than `'Fizz'` or `'Buzz'`.

Another way to determine if a number is divisible by 3 and 5 is to check if it is divisible by 15. This is because 15 is the least common multiple (LCM) of 3 and 5. You can either write the condition as `number % 3 == 0 and number % 5 == 0` or write the condition as `number % 15 == 0`.

The `range()` function tells `for` loops to go up to *but not including* their argument. The code `for i in range(10):` sets `i` to `0` through `9`, not `0` through `10`. You can specify two arguments to tell `range()` to start at a number besides `0`. The code `for i in range(1, 10):` sets `i` to `1` to `9`. In our exercise, you want to include the number in `upTo`, so add `1` to it: `range(1, upTo + 1)`

Python's `list()` function can take a `range` object returned from `range()` to produce a list of integers. For example, if you need a list of integers `1` to `10`, you can call `list(range(1, 11))`. If you need every even number between `150` up to and including `200`, you can call `list(range(150, 202, 2))`:

```
>>> list(range(1, 11))
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> list(range(150, 202, 2))
[150, 152, 154, 156, 158, 160, 162, 164, 166, 168, 170, 172, 174, 176, 178, 180,
182, 184, 186, 188, 190, 192, 194, 196, 198, 200]
```

Now try to write a solution based on the information in the previous sections. If you still have trouble solving this exercise, read the **Solution Template** section for additional hints.

Solution Template

Try to first write a solution from scratch. But if you have difficulty, you can use the following partial program as a starting place. Copy the following code from *https://invpy.com/fizzbuzz-template.py* and paste it into your code editor. Replace the underscores with code to make a working program:

```
def fizzBuzz(upTo):
    # Loop from 1 up to (and including) the upTo parameter:
    for number in range(1, ____):
        # If the loop number is divisible by 3 and 5, print 'FizzBuzz':
        if number % 3 == ____ and number % 5 == ____:
            ____(____, end=' ')
        # Otherwise the loop number is divisible by only 3, print 'Fizz':
        elif ____ % 3 == 0:
            ____(____, end=' ')
        # Otherwise the loop number is divisible by only 5, print 'Buzz':
        elif number % 5 ____ 0:
            ____(____, end=' ')
        # Otherwise, print the loop number:
        else:
            ____(____, end=' ')
```

The complete solution for this exercise is given in Appendix A and *https://invpy.com/fizzbuzz.py*. You can view each step of this program as it runs under a debugger at *https://invpy.com/fizzbuzz-debug/*.

EXERCISE #6: ORDINAL SUFFIX

```
ordinalSuffix(42)  ➔  '42nd'
```

While *cardinal numbers* refer to the size of a group of objects like "four apples" or "1,000 tickets", *ordinal numbers* refer to the place of an object in an ordered sequence like "first place" or "30th birthday". This exercise involves numbers but is more an exercise in processing text than doing math.

Exercise Description

In English, ordinal numerals have suffixes such as the "th" in "30th" or "nd" in "2nd". Write an `ordinalSuffix()` function with an integer parameter named `number` and returns a string of the number with its ordinal suffix. For example, `ordinalSuffix(42)` should return the string `'42nd'`.

You may use Python's `str()` function to convert the integer argument to a string. Python's `endswith()` string method could be useful for this exercise, but to maintain the challenge in this exercise, don't use it as part of your solution.

These Python `assert` statements stop the program if their condition is `False`. Copy them to the bottom of your solution program. Your solution is correct if the following `assert` statements' conditions are all `True`:

```
assert ordinalSuffix(0) == '0th'
assert ordinalSuffix(1) == '1st'
assert ordinalSuffix(2) == '2nd'
assert ordinalSuffix(3) == '3rd'
assert ordinalSuffix(4) == '4th'
assert ordinalSuffix(10) == '10th'
assert ordinalSuffix(11) == '11th'
assert ordinalSuffix(12) == '12th'
assert ordinalSuffix(13) == '13th'
assert ordinalSuffix(14) == '14th'
assert ordinalSuffix(101) == '101st'
```

Try to write a solution based on the information in this description. If you still have trouble solving this exercise, read the **Solution Design** and **Special Cases and Gotchas** sections for additional hints.

Prerequisite concepts: strings, `in` operator, slices, string concatenation

Solution Design

The ordinal suffix depends on the last two digits in the number:

- If the last two digits end with 11, 12, or 13 then the suffix is "th."
- If the number ends with 1, the suffix is "st."
- If the number ends with 2, the suffix is "nd."
- If the number ends with 3, the suffix is "rd."
- Every other number has a suffix of "th."

Our code becomes much easier to write if we call the `str()` function to convert the integer value in `number` to a string first. Then you can use negative indexes in Python to find the last digit in this string. You can store the string form in a variable named `numberStr`. Just as `numberStr[0]` and `numberStr[1]` evaluate to the first and second character in `numberStr`, `numberStr[-1]` and `numberStr[-2]` evaluate to the last and second to last character in `numberStr`. Therefore, you can use `numberStr[-1]` to check if the last digit is 1, 2, or 3.

To find the last two digits, you can use Python slices to get a substring from a string by specifying the start and end index of the slice. For example, `numberStr[0:3]` evaluates to the characters in `numberStr` from index `0` up to, but not including, index `3`. Enter the following in the interactive shell:

```
>>> numberStr = '41096'
>>> numberStr[0:3]
'410'
```

If `numberStr` were `'41096'`, `numberStr[0:3]` evaluates to `'410'`. If you leave out the first index, Python assumes you mean "the start of the string." For example, `numberStr[:3]` means the same thing as `numberStr[0:3]`. If you leave out the second index, Python assumes you mean "the end of the string." For example, `numberStr[3:]` means the same thing as `numberStr[3:len(numberStr)]` or `'96'`. Enter the following into the interactive shell:

```
>>> numberStr = '41096'
>>> numberStr[:3]
'410'
>>> numberStr[3:]
'96'
```

You can combine slices and negative indexes. For example, `numberStr[-2:]` means the slice starts from the second to last character in `numberStr` and continues to the end of the string. This is how `numberStr[-2:]` becomes the shorthand for "the last two characters in `numberStr`." You can use this to determine if the number ends with 11, 12, or 13. Enter the following into the interactive shell:

```
>>> numberStr = '41096'
>>> numberStr[-2:]
'96'
>>> numberStr = '987654321'
>>> numberStr[-2:]
'21'
```

If you need to compare a variable to multiple values, you can use the in keyword and put the values in a tuple. For example, instead of the lengthy condition `numberStr[-2:] == '11' or numberStr[-2:] == '12' or numberStr[-2:] == '13'`, you can have the equivalent and

more concise `numberStr[-2:] in ('11', '12', '13')`. Enter the following into the interactive shell:

```
>>> numberStr = '41096'
>>> numberStr[-2:] in ('11', '12', '13')
False
>>> numberStr = '512'
>>> numberStr[-2:] in ('11', '12', '13')
True
```

You'll use a similar expression in this exercise's solution.

Special Cases and Gotchas

Ensure that the number ends with 11, 12, or 13 *before* checking if it ends with 1, 2, or 3. Otherwise, you may end up programming your function to return `'213rd'` instead of `'213th'`. Also, notice that `'0'` has an ordinal suffix of "th": `'0th'`.

Now try to write a solution based on the information in the previous sections. If you still have trouble solving this exercise, read the **Solution Template** section for additional hints.

Solution Template

Try to first write a solution from scratch. But if you have difficulty, you can use the following partial program as a starting place. Copy the following code from *https://invpy.com/ordinalsuffix-template.py* and paste it into your code editor. Replace the underscores with code to make a working program:

```
def ordinalSuffix(number):
    numberStr = str(number)

    # 11, 12, and 13 have the suffix th:
    if numberStr[____:] in ('11', ____, ____):
        return numberStr + 'th'
    # Numbers that end with 1 have the suffix st:
    if numberStr[____] == '1':
        return numberStr + 'st'
    # Numbers that end with 2 have the suffix nd:
    if numberStr[____] == ____:
        return numberStr + 'nd'
    # Numbers that end with 3 have the suffix rd:
    if numberStr[____] == ____:
        return numberStr + 'rd'
    # All other numbers end with th:
    return ____ + ____
```

The complete solution for this exercise is given in Appendix A and *https://invpy.com/ordinalsuffix.py*. You can view each step of this program as it runs under a debugger at *https://invpy.com/ordinalsuffix-debug/*.

Another Solution Design

Instead of using the `str()` function to convert the number to a string and examining the digits at the end, you could leave it as an integer and determine the last digit with the `%` modulo operator.

Any number modulo 10 evaluates to the digit in the one's place of the number. You can use modulo 100 to get the last two digits. For example, enter the following into the interactive shell:

```
>>> 42 % 10
2
>>> 12345 % 10
5
>>> 12345 % 100
45
```

Using this information, write an alternative solution for this exercise using modulo. If you need additional hints, replace the underscores in the following solution template with code:

```
def ordinalSuffix(number):
    # 11, 12, and 13 have the suffix th:
    if ____ % 100 in (11, ____, ____):
        return str(number) + 'th'
    # Numbers that end with 1 have the suffix st:
    if ____ % 10 == ____:
        return str(number) + 'st'
    # Numbers that end with 2 have the suffix nd:
    if ____ % 10 == ____:
        return str(number) + 'nd'
    # Numbers that end with 3 have the suffix rd:
    if ____ % 10 == ____:
        return str(number) + 'rd'
    # All other numbers end with th:
    return ____(____) + ____
```

The complete solution for this exercise is given in Appendix A and *https://invpy.com/ordinalsuffix2.py*. You can view each step of this program as it runs under a debugger at *https://invpy.com/ordinalsuffix2-debug/*.

EXERCISE #7: ASCII TABLE

ASCII stands for *American Standard Code for Information Interchange*. Computers can only store numbers, so each letter, numeral, punctuation mark, and every other character is assigned a number called a *code point*. ASCII was a popular standard mapping of text characters to numbers. For example, `'Hello'` would be represented by 72, 101, 108, 108, 111. Specifically, computers only store the ones and zeros of binary numbers. These decimal numbers translate to 01001000, 01100101, 01101100, 01101100, 01101111 in binary. An ASCII table showed all the characters and their assigned ASCII number values.

However, ASCII is an old and somewhat limited standard: it doesn't have numbers assigned for Cyrillic or Chinese characters, for example. And it is an *American* standard: it has a code point for the dollar sign (code point 36) but not the British pound sign.

ASCII is no longer enough now that the internet has made global communication commonplace. The newer *Unicode* character set provides code points for every character and is what Python uses for its string values. Unicode's code points are backward compatible with ASCII, so we can still easily use Python to display an ASCII table.

In this exercise you'll learn how to use the `ord()` and `chr()` functions to translate between integers and text characters.

Exercise Description

Write a `printASCIITable()` function that displays the ASCII number and its corresponding text character, from 32 to 126. (These are called the *printable ASCII characters*.)

Your solution is correct if calling `printASCIITable()` displays output that looks like the following:

```
32
33 !
34 "
35 #
--more--
124 |
125 }
126 ~
```

Note that the character for ASCII value 32 is the space character, which is why it looks like nothing is next to 32 in the output.

Try to write a solution based on the information in this description. If you still have trouble solving this exercise, read the **Solution Design** and **Special Cases and Gotchas** sections for additional hints.

Prerequisite concepts: `for` loops, `range()` with two arguments, `chr()`

Solution Design

Python's built-in `chr()` function accepts an integer argument of a code point and returns the string of that code point's character. The `ord()` function does the opposite: accepting a string of a single character and returning its integer code point. The "ord" is short for "ordinal".

For example, enter the following into the interactive shell:

```
>>> ord('A')
65
>>> chr(65)
'A'
>>> ord('B')
66
>>> ord('!')
33
>>> 'Hello' + chr(33)
'Hello!'
```

The `printASCIITable()` function needs a `for` loop that starts at the integer `32` and goes up to `126`. Then, inside the loop, print the integer loop variable and what the `chr()` function returns for that integer loop variable.

Special Cases and Gotchas

The upper bound argument to the `range()` function doesn't include the number itself. This means that `for i in range(32, 126):` covers the range of integers from `32` up to, but not including, `126`. If you want to include `126` (which we do for this exercise), pass `127` as the second argument to `range()`.

Now try to write a solution based on the information in the previous sections. If you still have trouble solving this exercise, read the **Solution Template** section for additional hints.

Solution Template

Try to first write a solution from scratch. But if you have difficulty, you can use the following partial program as a starting place. Copy the following code from *https://invpy.com/asciitable-template.py* and paste it into your code editor. Replace the underscores with code to make a working program:

```
def printASCIITable():
    # Loop over integers 32 up to and including 126:
    for i in range(____, ____):
        # Print the integer and its ASCII text character:
        print(i, ____(i))

printASCIITable()
```

The complete solution for this exercise is given in Appendix A and *https://invpy.com/asciitable.py*.

You can view each step of this program as it runs under a debugger at *https://invpy.com/asciitable-debug/*.

Further Reading

If you'd like to learn more about Unicode, I recommend Ned Batchelder's "Pragmatic Unicode" blog post at *https://nedbatchelder.com/text/unipain.html*. He also gave a PyCon talk based on this blog post that you can watch at *https://youtu.be/sgHbC6udIqc*. Unicode is often seen as a complicated topic, but Ned breaks it down into understandable parts and what Python programmers need to know.

You can also see an ASCII table for yourself at *https://en.wikipedia.org/wiki/ASCII*.

EXERCISE #8: READ WRITE FILE

File I/O, or file input/output, allows your programs to read and write data to files on the hard drive. This exercise covers just the basics of writing text to a file with Python code and then reading the text from the file you just created. File I/O is an important technique because it allows your programs to save data so that work isn't lost when the program closes.

Exercise Description

You will write three functions for this exercise. First, write a `writeToFile()` function with two parameters for the filename of the file and the text to write into the file. Second, write an `appendToFile()` function, which is identical to `writeToFile()` except that the file opens in append mode instead of write mode. Finally, write a `readFromFile()` function with one parameter for the filename to open. This function returns the full text contents of the file as a string.

These Python instructions should generate the file and the `assert` statement checks that the content is correct:

```
writeToFile('greet.txt', 'Hello!\n')
appendToFile('greet.txt', 'Goodbye!\n')
assert readFromFile('greet.txt') == 'Hello!\nGoodbye!\n'
```

This code writes the text `'Hello!\n'` and `'Goodbye!\n'` to a file named *greet.txt*, then reads in this file's content to verify it is correct. You can delete the *greet.txt* file after running these instructions program.

Try to write a solution based on the information in this description. If you still have trouble solving this exercise, read the **Solution Design** and **Special Cases and Gotchas** sections for additional hints.

Prerequisite concepts: text file reading and writing

Solution Design

Python's built-in `open()` function returns a file object that you can use to read or write text from and to the file. You should call `open()` with the `with` statement, like `with open(filename, mode) as fileObj:`.

The valid arguments for the `mode` parameter are `'r'` to read from text files, `'w'` to write to text files, and `'a'` to append (that is, write at the end of) to text files. If no mode argument is given, then the function defaults to read mode. Opening the file in write or append mode lets you call

`fileObj.write('text goes here')`, which allows you to pass a string of text to write to the file. Otherwise, in read mode calling `fileObj.read()` returns the file's contents as a string.

Special Cases and Gotchas

If you open a file in write mode and the file doesn't exist, then a blank file is created. If the file does exist, it is overwritten with a blank file. Append mode, as opposed to write mode, adds text to the end of the file without destroying the original content.

Now try to write a solution based on the information in the previous sections. If you still have trouble solving this exercise, read the **Solution Template** section for additional hints.

Solution Template

Try to first write a solution from scratch. But if you have difficulty, you can use the following partial program as a starting place. Copy the following code from *https://invpy.com/readwritefile-template.py* and paste it into your code editor. Replace the underscores with code to make a working program:

```
def writeToFile(filename, text):
    # Open the file in write mode:
    with open(filename, ____) as fileObj:
        # Write the text to the file:
        ____.write(text)

def appendToFile(filename, text):
    # Open the file in append mode:
    ____ open(filename, ____) as fileObj:
        # Write the text to the end of the file:
        ____.write(____)

def readFromFile(filename):
    # Open the file in read mode:
    ____ ____(filename) as fileObj:
        # Read all of the text in the file and return it as a string:
        return ____.read()
```

The complete solution for this exercise is given in Appendix A and *https://invpy.com/readwritefile.py*. You can view each step of this program as it runs under a debugger at *https://invpy.com/readwritefile-debug/*.

Further Reading

You can learn about reading and writing files in Chapters 9 and 10 of my book, *Automate the Boring Stuff with Python* online at *https://automatetheboringstuff.com/2e/*.

EXERCISE #9: CHESS SQUARE COLOR

```
getChessSquareColor(8, 8)  →  'white'
getChessSquareColor(2, 1)  →  'black'
```

A chess board has a checker-pattern of white and black tiles. In this program, you'll determine a pattern to the color of the squares based on their column and row. This program challenges you to take a real-world object such as a chess board, determine the patterns behind its design, and translate that into Python code.

Exercise Description

Write a `getChessSquareColor()` function that has parameters `column` and `row`. The function either returns `'black'` or `'white'` depending on the color at the specified `column` and `row`. Chess boards are 8 x 8 spaces in size, and the columns and rows in this program begin at `0` and end at `7` like in Figure 9-1. If the arguments for `column` or `row` are outside the `0` to `7` range, the function returns a blank string.

Note that chess boards always have a white square in the top left corner.

Figure 9-1: A chess board with labeled columns and rows.

These Python `assert` statements stop the program if their condition is `False`. Copy them to

the bottom of your solution program. Your solution is correct if the following `assert` statements' conditions are all `True`:

```
assert getChessSquareColor(1, 1) == 'white'
assert getChessSquareColor(2, 1) == 'black'
assert getChessSquareColor(1, 2) == 'black'
assert getChessSquareColor(8, 8) == 'white'
assert getChessSquareColor(0, 8) == ''
assert getChessSquareColor(2, 9) == ''
```

Try to write a solution based on the information in this description. If you still have trouble solving this exercise, read the **Solution Design** and **Special Cases and Gotchas** sections for additional hints.

Prerequisite concepts: Boolean operators, modulo operator

Solution Design

There is a pattern to the colors of a chess board. If the column and row are both even or both odd, then the space is white. If one is odd and the other is even, the space is black. Exercise #3, "Odd & Even" shows how the `%` modulo operator determines if a number (such as the one in the `column` and `row` parameter) is even or odd. Use this to write a condition that determines if the oddness or evenness of the column and row match.

Special Cases and Gotchas

The function should first check whether the column or row argument is outside the `0` to `7` range of valid values. If so, the function should immediately return a blank string.

Now try to write a solution based on the information in the previous sections. If you still have trouble solving this exercise, read the **Solution Template** section for additional hints.

Solution Template

Try to first write a solution from scratch. But if you have difficulty, you can use the following partial program as a starting place. Copy the following code from *https://invpy.com/chesscolor-template.py* and paste it into your code editor. Replace the underscores with code to make a working program:

```
def getChessSquareColor(column, row):
    # If the column and row is out of bounds, return a blank string:
    if column ____ or ____ > 8 or ____ < 1 or row ____:
        return ''

    # If the even/oddness of the column and row match, return 'white':
    if ____ % 2 == row % ____:
        return 'white'
    # If they don't match, then return 'black':
    else:
        return 'black'
```

The complete solution for this exercise is given in Appendix A and *https://invpy.com/chesscolor.py*. You can view each step of this program as it runs under a debugger at *https://invpy.com/chesscolor-debug/*.

EXERCISE #10: FIND AND REPLACE

```
findAndReplace('The fox', 'fox', 'dog')  ➔  'The dog'
```

Find-and-replace is a standard feature in text editors, IDEs, and word processor software. Even the Python language comes with a `replace()` string method since programs often use it. In this exercise, you'll reimplement this common string operation.

Exercise Description

Write a `findAndReplace()` function that has three parameters: `text` is the string with text to be replaced, `oldText` is the text to be replaced, and `newText` is the replacement text. Keep in mind that this function must be case-sensitive: if you are replacing `'dog'` with `'fox'`, then the `'DOG'` in `'MY DOG JONESY'` won't be replaced.

These Python `assert` statements stop the program if their condition is `False`. Copy them to the bottom of your solution program. Your solution is correct if the following `assert` statements' conditions are all `True`:

```
assert findAndReplace('The fox', 'fox', 'dog') == 'The dog'
assert findAndReplace('fox', 'fox', 'dog') == 'dog'
assert findAndReplace('Firefox', 'fox', 'dog') == 'Firedog'
assert findAndReplace('foxfox', 'fox', 'dog') == 'dogdog'
assert findAndReplace('The Fox and fox.', 'fox', 'dog') == 'The Fox and dog.'
```

Try to write a solution based on the information in this description. If you still have trouble solving this exercise, read the **Solution Design** and **Special Cases and Gotchas** sections for additional hints.

Prerequisite concepts: slices, indexes, `len()`, augmented assignment operator

Solution Design

While Python comes with several string methods, such as `replace()`, `find()`, and `index()`, that could do this exercise for us, we'll do the finding and replacing on our own.

At the beginning of the function, create a `replacedText` variable to hold the text with replacements. It starts as a blank string. We'll write code that copies the text in the `text` parameter to `replacedText`, except for where it finds instances of `oldText`, in which case `newText` is copied

to `replacedText`.

Create a `while` loop starts a variable named `i` at `0` and then keeps looping until `i` reaches the length of the `text` string argument. This `i` variable points to an index in the `text` string. The code inside the loop increases `i` by the length of `oldText` if it has found an instance of `oldText` in `text`. Otherwise the code inside the loop increases `i` by `1` so it can examine the next character in `text`.

The code inside the loop can examine the slice `text[i:i + len(oldText)]` to see if it matches `oldText`. In that case, we append `newText` to `replacedText` and increase `i` by `1`. If not, we append just the `text[i]` character to `replacedText` and increase `i` by `len(oldText)`. By the time `i` reaches the end of `text`, the `replacedText` variable contains the finished string.

Figure 10-1 shows each step of this process if we were to call `findAndReplace('A fox.', 'fox', 'dog')`.

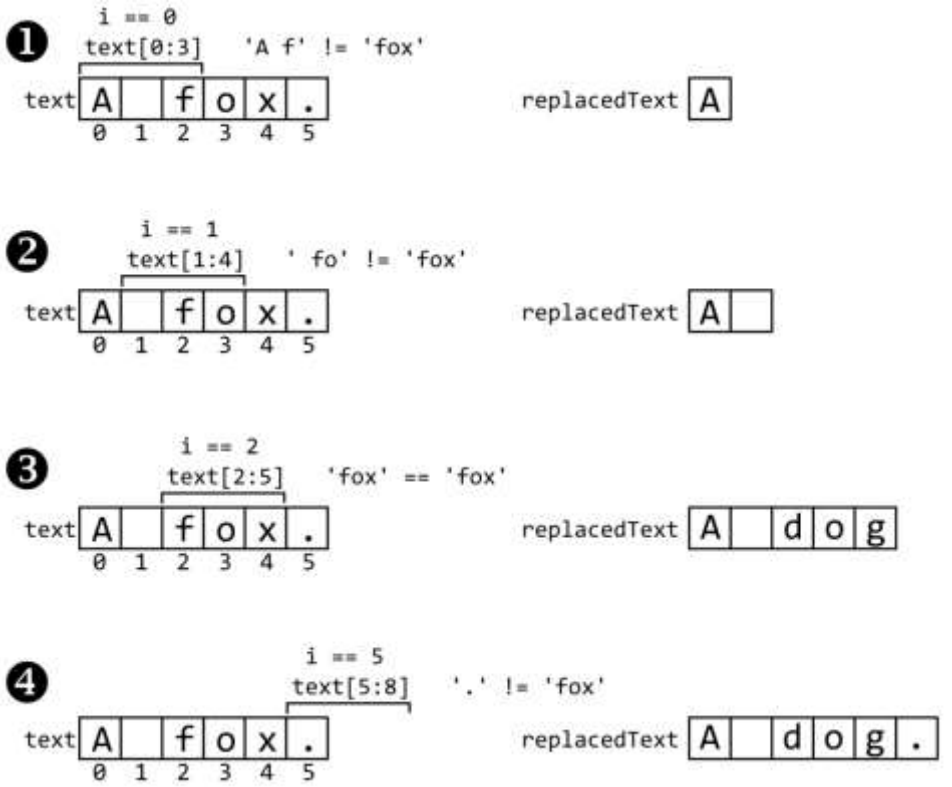


Figure 10-1: Replacing `'fox'` with `'dog'` in `'A fox.'`

Python has *augmented assignment operators* such as `+=` and `*=`. These are shortcuts for assignment statements that modify the value in a variable. Python has augmented assignment operators for several operators:

Augmented Assignment	**Equivalent To**
`someVariable += 42`	`someVariable = someVariable + 42`
`someVariable -= 42`	`someVariable = someVariable - 42`

`someVariable *= 42`	`someVariable = someVariable * 42`
`someVariable /= 42`	`someVariable = someVariable / 42`
`someVariable %= 42`	`someVariable = someVariable % 42`
`someVariable //= 42`	`someVariable = someVariable // 42`
`someVariable **= 42`	`someVariable = someVariable ** 42`

For example, instead of typing `someVariable = someVariable + 42`, you could have the equivalent `someVariable += 42` as a more concise form. Python's augmented assignment operators include `+=`, `-=`, `*=`, `/=`, and `%=`.

The solution I give for this exercise includes the `+=` augmented assignment operator.

Special Cases and Gotchas

Using an out-of-bounds index on a string results an error message: `'Hello'[9999]` causes and `IndexError: string index out of range` error. However, this doesn't apply to slices. Running `'Hello'[1:9999]` results in `'ello'` rather than an error message, even though `9999` is greater than the largest index of the string, `4`.

Keep this in mind when comparing slices with strings. For example, the slice `text[5:8]` might evaluate to a string of 3 characters because 8 - 5 is 3. But if this slice's indexes are towards the end of the text string, it could evaluate to a string of 2, 1, or 0 characters. So think of the slice `text[5:8]` as being *at most* 3 characters long.

For example, enter the following into the interactive shell for an example:

```
>>> 'elephant'[5:8]
'ant'
>>> 'gazelle'[5:8]
'le'
>>> 'turtle'[5:8]
'e'
>>> 'moose'[5:8]
''
```

Now try to write a solution based on the information in the previous sections. If you still have trouble solving this exercise, read the **Solution Template** section for additional hints.

Solution Template

Try to first write a solution from scratch. But if you have difficulty, you can use the following partial program as a starting place. Copy the following code from *https://invpy.com/findandreplace-template.py* and paste it into your code editor. Replace the underscores with code to make a working program:

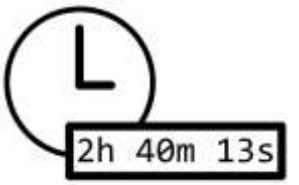
```
def findAndReplace(text, oldText, newText):
    replacedText = ____
```

```
    i = ____
    while i < len(____):
        # If index i in text is the start of the oldText pattern, add
        # the replacement text:
        if text[i:i + len(____)] == oldText:
            # Add the replacement text:
            replacedText += ____
            # Increment i by the length of oldText:
            i += len(____)
        # Otherwise, add the characters at text[i] and increment i by 1:
        else:
            replacedText += ____[i]
            i += ____
    return replacedText
```

The complete solution for this exercise is given in Appendix A and *https://invpy.com/findandreplace.py*. You can view each step of this program as it runs under a debugger at *https://invpy.com/ findandreplace-debug/*.

EXERCISE #11: HOURS, MINUTES, SECONDS

```
getHoursMinutesSeconds(90)  →  '1m 30s'
```



Websites often use relative timestamps such as "3 days ago" or "about 3h ago" so the user doesn't need to compare an absolute timestamp to the current time. In this exercise, you write a function that converts a number of seconds into a string with the number of hours, minutes, and seconds.

Exercise Description

Write a `getHoursMinutesSeconds()` function that has a `totalSeconds` parameter. The argument for this parameter will be the number of seconds to be translated into the number of hours, minutes, and seconds. If the amount for the hours, minutes, or seconds is zero, don't show it: the function should return `'10m'` rather than `'0h 10m 0s'`. The only exception is that `getHoursMinutesSeconds(0)` should return `'0s'`.

These Python `assert` statements stop the program if their condition is `False`. Copy them to the bottom of your solution program. Your solution is correct if the following `assert` statements' conditions are all `True`:

```
assert getHoursMinutesSeconds(30) == '30s'
assert getHoursMinutesSeconds(60) == '1m'
assert getHoursMinutesSeconds(90) == '1m 30s'
assert getHoursMinutesSeconds(3600) == '1h'
assert getHoursMinutesSeconds(3601) == '1h 1s'
assert getHoursMinutesSeconds(3661) == '1h 1m 1s'
assert getHoursMinutesSeconds(90042) == '25h 42s'
assert getHoursMinutesSeconds(0) == '0s'
```

For an additional challenge, break up 24 hour periods into days with a "d" suffix. For example, `getHoursMinutesSeconds(90042)` would return `'1d 1h 42s'`.

Try to write a solution based on the information in this description. If you still have trouble solving this exercise, read the **Solution Design** and **Special Cases and Gotchas** sections for additional hints.

Prerequisite concepts: `join()`, `append()`, lists, string concatenation, `while` loops

Solution Design

One hour is 3600 seconds, one minute is 60 seconds, and one second is… 1 second. Our solution can have variables that track the number of hours, minutes, and seconds in the final result with variables `hours`, `minutes`, and `seconds`. The code subtracts `3600` from the `totalSeconds` while increasing `hours` by `1`. Then the code subtracts `60` from `totalSeconds` while increasing `minutes` by 1. The `seconds` variable can then be set to whatever remains in `totalSeconds`.

After calculating hours, minutes, and seconds, you should convert those integer amounts into strings with respective `'h'`, `'m'`, and `'s'` suffixes like `'1h'` or `'30s'`. Then you can append them to a list that initially starts as empty. Then the `join()` list method can join these strings together with a single space separating them: from the list `['1h', '30s']` to the string `'1h 30s'`. For example, enter the following into the interactive shell:

```
>>> hms = ['1h', '30s']
>>> ' '.join(hms)
'1h 30s'
```

The function then returns this space-separated string.

If you're unfamiliar with the `join()` string method, you can enter `help(str.join)` into the interactive shell to view its documentation, or do an internet search for "python join".

Special Cases and Gotchas

The first special case your function should check for is if the `totalSeconds` parameter is set to 0. In this case, the function can immediately return `'0s'`.

The final result string shouldn't include hours, minutes, or seconds if the amount of those is zero. For example, your program should return `'1h 12s'` but not `'1h 0m 12s'`.

Also, it's important to subtract the larger amounts first. Otherwise, you would, for example, increase `hours` by `0` and minutes by `120` instead of increase `hours` by `2` and `minutes` by `0`.

Now try to write a solution based on the information in the previous sections. If you still have trouble solving this exercise, read the **Solution Template** section for additional hints.

Solution Template

Try to first write a solution from scratch. But if you have difficulty, you can use the following partial program as a starting place. Copy the following code from *https://inventwithpython.com/hoursminutesseconds-template.py* and paste it into your code editor. Replace the underscores with code to make a working program:

```
def getHoursMinutesSeconds(totalSeconds):
    # If totalSeconds is 0, just return '0s':
    if totalSeconds == ____:
        return ____

    # Set hours to 0, then add an hour for every 3600 seconds removed from
    # totalSeconds until totalSeconds is less than 3600:
    hours = 0
```

```
    while totalSeconds ____ 3600:
        hours += ____
        totalSeconds -= ____

    # Set minutes to 0, then add a minute for every 60 seconds removed from
    # totalSeconds until totalSeconds is less than 60:
    minutes = 0
    while totalSeconds >= ____:
        minutes += 1
        totalSeconds -= ____

    # Set seconds to the remaining totalSeconds value:
    seconds = ____

    # Create an hms list that contains the string hour/minute/second amounts:
    hms = []
    # If there are one or more hours, add the amount with an 'h' suffix:
    if hours > ____:
        ____.append(str(____) + 'h')
    # If there are one or more minutes, add the amount with an 'm' suffix:
    if minutes > 0:
        hms.append(____(minutes) + 'm')
    # If there are one or more seconds, add the amount with an 's' suffix:
    if seconds > 0:
        hms.append(str(seconds) + ____)

    # Join the hour/minute/second strings with a space in between them:
    return ' '.join(____)
```

The complete solution for this exercise is given in Appendix A and *https://inventwithpython.com/hoursminutesseconds.py*. You can view each step of this program as it runs under a debugger at *https://inventwithpython.com/hoursminutesseconds-debug/*.

Alternate Solution Design

Alternatively, instead of subtractions in a loop you can use Python's `//` *integer division operator* to see how many 3600 amounts (for hours) or 60 amounts (for minutes) fit into `totalSeconds`. To get the remaining amount in `totalSeconds` after removing the seconds accounted for hours and minutes, you can use Python's `%` modulo operator.

For example, if `totalSeconds` were `10000`, then `10000 // 3600` evaluates to `2`, telling you that there are two hours in 10,000 seconds. Then `10000 % 3600` evaluates to `2800`, telling you there are 2,800 seconds left over after removing the two hours from 10,000 seconds. You would then run `2800 // 60` to find the number of minutes in 2,800 seconds and `2800 % 60` to find the number of seconds remaining after removing those minutes.

Alternate Solution Template

Try to first write a solution from scratch. But if you have difficulty, you can use the following partial program as a starting place. Copy the following code from *https://inventwithpython.com/hoursminutesseconds2-template.py* and paste it into your code editor. Replace the underscores with code to make a working program:

```
def getHoursMinutesSeconds(totalSeconds):
```

```
    # If totalSeconds is 0, just return '0s':
    if totalSeconds == ____:
        return ____

    # Set hours to how many times 3600 seconds can divide
    # totalSeconds. Then set totalSeconds to the remainder:
    if totalSeconds >= 3600:
        hours = totalSeconds // 3600
        totalSeconds = totalSeconds % 3600
    else:
        hours = 0

    # Set minutes to how many times 60 seconds can divide
    # totalSeconds. Then set totalSeconds to the remainder:
    if totalSeconds >= 60:
        minutes = totalSeconds // 60
        totalSeconds = totalSeconds % 60
    else:
        minutes = 0

    # Set seconds to the remaining totalSeconds value:
    seconds = ____

    # Create an hms list that contains the string hour/minute/second amounts:
    hms = []
    # If there are one or more hours, add the amount with an 'h' suffix:
    if hours > ____:
        ____.append(str(____) + 'h')
    # If there are one or more minutes, add the amount with an 'm' suffix:
    if minutes > 0:
        hms.append(____(minutes) + 'm')
    # If there are one or more seconds, add the amount with an 's' suffix:
    if seconds > 0:
        hms.append(str(seconds) + ____)

    # Join the hour/minute/second strings with a space in between them:
    return ' '.join(____)
```

The complete solution for this exercise is given in Appendix A and *https://invpy.com/hoursminutesseconds2.py*. You can view each step of this program as it runs under a debugger at *https://invpy.com/hoursminutesseconds2-debug/*.

EXERCISE #12: SMALLEST & BIGGEST

```
getSmallest([28, 25, 42, 2, 28])  →  2
```

Python's built-in `min()` and `max()` functions return the smallest and biggest numbers in a list of numbers passed, respectively. In this exercise, you'll reimplement the behavior of these functions.

This is the sort of problem that is trivial for a human to solve by hand if the list of numbers is short. However, if the list contains thousands, millions, or billions of numbers, you'll need to program a computer to perform the calculation.

Exercise Description

Write a `getSmallest()` function that has a `numbers` parameter. The `numbers` parameter will be a list of integer and floating-point number values. The function returns the smallest value in the list. If the list is empty, the function should return `None`. Since this function replicates Python's `min()` function, your solution shouldn't use it.

These Python `assert` statements stop the program if their condition is `False`. Copy them to the bottom of your solution program. Your solution is correct if the following `assert` statements' conditions are all `True`:

```
assert getSmallest([1, 2, 3]) == 1
assert getSmallest([3, 2, 1]) == 1
assert getSmallest([28, 25, 42, 2, 28]) == 2
assert getSmallest([1]) == 1
assert getSmallest([]) == None
```

Try to write a solution based on the information in this description. If you still have trouble solving this exercise, read the **Solution Design** and **Special Cases and Gotchas** sections for additional hints.

When you are done with this exercise, write a `getBiggest()` function which returns the biggest number instead of the smallest number.

Prerequisite concepts: `len()`, `for` loops, lists, `None` value

Solution Design

Think about how you would solve this problem without a computer, given a list of numbers written on paper. You would use the first number as the smallest number, and then read every number after it. If the next number is smallest than the smallest number you've seen so far, it becomes the new smallest number. Let's look at a small example. Figure 12-1 shows how looping over the list `[28, 25, 42, 2, 28]` would affect the contents of a variable named `smallest` that tracks the smallest number seen so far.

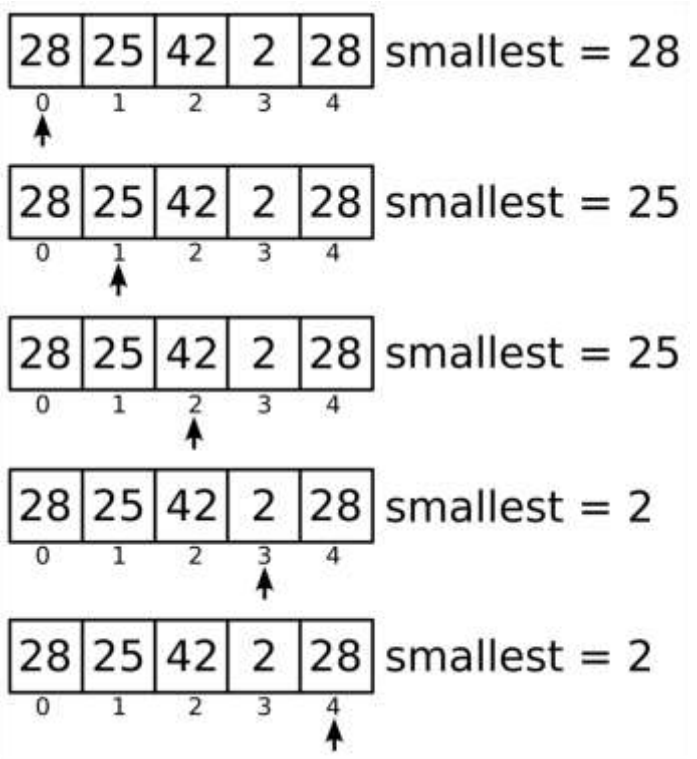


Figure 12-1: The value in `smallest` contains the smallest integer found so far as the `for` loop iterates over the list `[28, 25, 42, 2, 28]`.

Creates a variable named `smallest` to track the smallest value found so far and set it to the first value in the list to start. Then have a `for` loop that loops over every number in the list from left to right, and if the number is less than the current value in `smallest`, it becomes the new value in `smallest`. After the loop finishes, the function returns the smallest value.

Special Cases and Gotchas

The function should first check if the list is empty. In that case, return `None`. And by starting the `smallest` variable to the first number in a non-empty `numbers` list, you can guarantee that `smallest` is always initialized to a value from the list.

Now try to write a solution based on the information in the previous sections. If you still have trouble solving this exercise, read the **Solution Template** section for additional hints.

Solution Template

Try to first write a solution from scratch. But if you have difficulty, you can use the following partial program as a starting place. Copy the following code from *https://invpy.com/smallest-template.py* and paste it into your code editor. Replace the underscores with code to make a working program:

```
def getSmallest(numbers):
    # If the numbers list is empty, return None:
    if len(____) == ____:
        return None

    # Create a variable that tracks the smallest value so far, and start
    # it off a the first value in the list:
    smallest = numbers[____]
    # Loop over each number in the numbers list:
    for number in ____:
        # If the number is smaller than the current smallest value, make
        # it the new smallest value:
        if ____ < smallest:
            ____ = number
    # Return the smallest value found:
    ____ smallest
```

The complete solution for this exercise is given in Appendix A and *https://invpy.com/smallest.py*. You can view each step of this program as it runs under a debugger at *https://invpy.com/smallest-debug/*.

Further Reading

The benefit of writing a computer program to do a simple task like finding the smallest number in a list is that a computer can process a list of millions of numbers in seconds. We can simulate this by having the computer generate one million random numbers in a list, and then pass that list to our `getSmallest()` function. On my computer, this program takes a few seconds, and most of that time is spent displaying the million numbers on the screen.

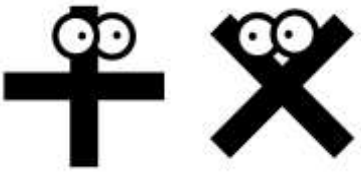
Write the following program and save it in a file named *testsmallest.py*. Run it from the same folder as your *smallest.py* file so that it can import it as a module:

```
import random, smallest

numbers = []
for i in range(1000000):
    numbers.append(random.randint(1, 1000000000))
print('Numbers:', numbers)
print('Smallest number is', smallest.getSmallest(numbers))
```

When run, this program displays the million numbers between 1 and 1,000,000,000 it generated, along with the smallest number in that list.

EXERCISE #13: SUM & PRODUCT

```
    calculateSum([2, 4, 6, 8, 10])  →  30
calculateProduct([2, 4, 6, 8, 10])  →  3840
```



Python's built-in `sum()` function returns the sum of the list of numbers passed for its argument. In this exercise, you'll reimplement this behavior for your `calculateSum()` function and also create a `calculateProduct()` function.

Exercise Description

Write two functions named `calculateSum()` and `calculateProduct()`. They both have a parameter named `numbers`, which will be a list of integer or floating-point values. The `calculateSum()` function adds these numbers and returns the sum while the `calculateProduct()` function multiplies these numbers and returns the product. If the list passed to `calculateSum()` is empty, the function returns `0`. If the list passed to `calculateProduct()` is empty, the function returns `1`. Since this function replicates Python's `sum()` function, your solution shouldn't call.

These Python `assert` statements stop the program if their condition is `False`. Copy them to the bottom of your solution program. Your solution is correct if the following `assert` statements' conditions are all `True`:

```
assert calculateSum([]) == 0
assert calculateSum([2, 4, 6, 8, 10]) == 30
assert calculateProduct([]) == 1
assert calculateProduct([2, 4, 6, 8, 10]) == 3840
```

Try to write a solution based on the information in this description. If you still have trouble solving this exercise, read the **Solution Design** and **Special Cases and Gotchas** sections for additional hints.

Prerequisite concepts: lists, indexes, `for` loops, augmented assignment operator

Solution Design

For calculating a sum in `calculateSum()`, create a variable named `result` to store the

running sum result. This variable starts at `0`, and while the program loops over each number in the numbers parameter, each number is added to this running sum. The same is done in `calculateProduct()`, except the `result` variable starts at `1` and is multiplied by each number.

Special Cases and Gotchas

You might be tempted to use a variable named `sum` to store the running sum. However, in Python this name is already used for the built-in `sum()` function. Reusing this name for a local variable means we'd lose the ability to call the original `sum()` function for the rest of the function's code because we have overwritten `sum` to the value we put in it rather than Python's `sum()` function. You want to avoid reusing the names of Python's built-in functions by overwriting them, as it could cause odd bugs in your code. Some commonly overwritten Python names are `all`, `any`, `date`, `email`, `file`, `format`, `hash`, `id`, `input`, `list`, `min`, `max`, `object`, `open`, `random`, `set`, `str`, `sum`, `test`, and `type`. Don't use these names for your own variables.

Also, the sum of an empty list of numbers should be `0`. The product of an empty list of numbers should be `1`. When calculating the product of several numbers, begin the running product result at `1` and not `0`. If you start it at `0`, all multiplications result in `0` and your `calculateProduct()` function will return `0` every time. In mathematics, the numbers 0 and 1 are called the *additive identity* and *multiplicative identity*, respectively. Adding the additive identity (0) to a number results in that same number, while multiplying the multiplicative identity (1) with a number results in that same number. For example, 12 + 0 is just 12 and 12 × 1 is just 12 too.

Now try to write a solution based on the information in the previous sections. If you still have trouble solving this exercise, read the **Solution Template** section for additional hints.

Solution Template

Try to first write a solution from scratch. But if you have difficulty, you can use the following partial program as a starting place. Copy the following code from *https://invpy.com/sumproduct-template.py* and paste it into your code editor. Replace the underscores with code to make a working program:

```
def calculateSum(numbers):
    # Start the sum result at 0:
    result = ____
    # Loop over all the numbers in the numbers parameter, and add them
    # to the running sum result:
    for number in ____:
        result += ____
    # Return the final sum result:
    return ____

def calculateProduct(numbers):
    # Start the product result at 1:
    result = ____
    # Loop over all the numbers in the numbers parameter, and multiply
    # them by the running product result:
    for number in ____:
        result ____ number
    # Return the final product result:
    return ____
```

The complete solution for this exercise is given in Appendix A and *https://invpy.com/sumproduct.py*.

You can view each step of this program as it runs under a debugger at *https://inventwithpython.com/sumproduct-debug/*.

Further Reading

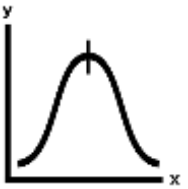
Just like in Exercise #12 "Smallest & Biggest" we can generate a list of one million numbers, and then calculate the sum and product of these numbers. Multiplying one million numbers creates an astronomically large number, so we'll limit our test to a mere ten thousand numbers. Write the following program and save it in a file named *testsumproduct.py*. Run it from the same folder as your *sumproduct.py* file so that it can import it as a module:

```
import random, sumproduct

numbers = []
for i in range(10000):
    numbers.append(random.randint(1, 1000000000))
print('Numbers:', numbers)
print('    Sum is', sumproduct.calculateSum(numbers))
print('Product is', sumproduct.calculateProduct(numbers))
```

When run, this program displays the million numbers between 1 and 1,000,000,000 it generated, along with the sum and product of the numbers in that list.

EXERCISE #14: AVERAGE

```
average([12, 20, 37])  →  23
```



Averages are an essential statistical tool, and computers make it easy to calculate the average of millions or billions of numbers. The *average* is the sum of a set of the numbers divided by the amount of numbers. For example, the average of 12, 1, and 5 is 6, because 12 + 1 + 5 is 18 and 18 / 3 is 6. This and the following two exercises challenge you to make Python solve these statistics calculations.

Exercise Description

Write an `average()` function that has a `numbers` parameter. This function returns the statistical average of the list of integer and floating-point numbers passed to the function. While Python's built-in `sum()` function can help you solve this exercise, try writing the solution without using it.

Passing an empty list to `average()` should cause it to return `None`.

These Python `assert` statements stop the program if their condition is `False`. Copy them to the bottom of your solution program. Your solution is correct if the following `assert` statements' conditions are all `True`:

```
assert average([1, 2, 3]) == 2
assert average([1, 2, 3, 1, 2, 3, 1, 2, 3]) == 2
assert average([12, 20, 37]) == 23
assert average([0, 0, 0, 0, 0]) == 0
import random
random.seed(42)
testData = [1, 2, 3, 1, 2, 3, 1, 2, 3]
for i in range(1000):
    random.shuffle(testData)
    assert average(testData) == 2
```

Shuffling the order of the numbers should not affect the average. The `for` loop does 1,000 such random shuffles to thoroughly check that this fact remains true. For an explanation of the `random.seed()` function, see the Further Reading section of Exercise #19, "Password Generator".

Try to write a solution based on the information in this description. If you still have trouble solving this exercise, read the **Solution Design and Special Cases and Gotchas** sections for additional hints.

Prerequisite concepts: `len()`, `for` loops, augmented assignment operators

Solution Design

Create a variable that holds the sum total of the numbers in the `numbers` list. Then write a `for` loop to go over all the numbers in the list, adding them to this total. This is identical to the `calculateSum()` function in Exercise #13 "Sum & Product." After the loop, return the total divided by the amount of numbers in the `numbers` list. You can use Python's built-in `len()` function for this.

While you shouldn't use Python's built-in `sum()` function, you can import your *sumproduct.py* program from Exercise #13, "Sum & Product" for its `calculateSum()` function. After completing Exercise #13, make sure *sumproduct.py* is in the same folder as *average.py*. Then use `import sumproduct` to import the module and `sumproduct.calculateSum()` to call the function.

Special Cases and Gotchas

If the `numbers` parameter is an empty list, the function should return `None`. Therefore, you should put the code that checks this at the start of the function.

Now try to write a solution based on the information in the previous sections. If you still have trouble solving this exercise, read the **Solution Template** section for additional hints.

Solution Template

Try to first write a solution from scratch. But if you have difficulty, you can use the following partial program as a starting place. Copy the following code from *https://inventwithpython.com/average-template.py* and paste it into your code editor. Replace the underscores with code to make a working program:

```
def average(numbers):
    # Special case: If the numbers list is empty, return None:
    if len(____) == ____:
        return ____

    # Start the total at 0:
    total = ____

    # Loop over each number in numbers:
    for number in ____:
        # Add the number to the total:
        total ____ number

    # Get the average by dividing the total by how many numbers there are:
    return ____ / ____(numbers)
```

The complete solution for this exercise is given in Appendix A and *https://inventwithpython.com/average.py*. You can view each step of this program as it runs under a debugger at *https://inventwithpython.com/average-debug/*.

EXERCISE #15: MEDIAN

```
median([3, 7, 10, 4, 1, 6, 9, 2, 8])  →  6
```

If you put a list of numbers into sorted order, the *median* number is the number at the halfway point. Outliers can cause the statistical average to be much higher or smaller than the majority of numbers, so that the median number may give you a better idea of the characteristics of the numbers in the list. This, the previous, and the next exercise challenge you to make Python solve these statistics calculations.

Exercise Description

Write a `median()` function that has a `numbers` parameter. This function returns the statistical median of the `numbers` list. The median of an odd-length list is the number in the middlemost number when the list is in sorted order. If the list has an even length, the median is the average of the two middlemost numbers when the list is in sorted order. Feel free to use Python's built-in `sort()` method to sort the `numbers` list.

Passing an empty list to `median()` should cause it to return `None`.

These Python `assert` statements stop the program if their condition is `False`. Copy them to the bottom of your solution program. Your solution is correct if the following `assert` statements' conditions are all `True`:

```
assert median([]) == None
assert median([1, 2, 3]) == 2
assert median([3, 7, 10, 4, 1, 9, 6, 5, 2, 8]) == 5.5
assert median([3, 7, 10, 4, 1, 9, 6, 2, 8]) == 6
import random
random.seed(42)
testData = [3, 7, 10, 4, 1, 9, 6, 2, 8]
for i in range(1000):
    random.shuffle(testData)
    assert median(testData) == 6
```

Shuffling the order of the numbers should not affect the median. The `for` loop does 1,000 such random shuffles to thoroughly check that this fact remains true. For an explanation of the `random.seed()` function, see the **Further Reading** section of Exercise #19, "Password Generator".

Try to write a solution based on the information in this description. If you still have trouble

solving this exercise, read the **Solution Design and Special Cases and Gotchas** sections for additional hints.

Prerequisite concepts: `len()`, `for` loops, augmented assignment operators, integer division, modulo operator, indexes

Solution Design

First, check if the `numbers` list is empty and, if so, return `None`. Next, you must sort them by calling the `sort()` list method. Then calculate the middle index by integer dividing the list length by `2`. Integer division does normal division and then rounds the result down to the next lowest integer. Python's integer division operator is `//`. So, for example, while the expression `5 / 2` evaluates to `2.5`, the expression `5 // 2` evaluates to `2`.

Next, figure out if the list length is odd or even. If odd, the median number is at the middle index. Let's think about a few examples to make sure this is correct:

- If the list length is `3`, the indexes range from `0` to `2` and the middle index is `3 // 2` or `1`. And `1` is the middle of `0` to `2`.
- If the list length is `5`, the indexes range from `0` to `4` and the middle index is `5 // 2` or `2`. And `2` is the middle of `0` to `4`.
- If the list length is `9`, the indexes range from `0` to `8` and the middle index is `9 // 2` or `4`. And `4` is the middle of `0` to `8`.

These seem correct. If the list length is even, we need to calculate the average of the two middle numbers. The indexes for these are the middle index and the middle index minus 1. Let's think about a few examples to make sure this is correct:

- If the list length is `4`, the indexes range from `0` to `3` and the middle indexes are `4 // 2` and `4 // 2 - 1`, or `2` and `1`. And `2` and `1` are the middle of `0` to `3`.
- If the list length is `6`, the indexes range from `0` to `5` and the middle indexes are `6 // 2` and `6 // 2 - 1`, or `3` and `2`. And `3` and `2` are the middle of `0` to `5`.
- If the list length is `10`, the indexes range from `0` to `9` and the middle indexes are `10 // 2` and `10 // 2 - 1`, or `5` and `4`. And `5` and `4` are the middle of `0` to `9`.

These seem correct too. Even-length lists have the additional step that the median is the average of two numbers, so add them together and divide by two.

Special Cases and Gotchas

If the `numbers` parameter is an empty list, the function should return `None`. Therefore, you should put the code that checks this at the start of the function.

Now try to write a solution based on the information in the previous sections. If you still have trouble solving this exercise, read the **Solution Template** section for additional hints.

Solution Template

Try to first write a solution from scratch. But if you have difficulty, you can use the following partial program as a starting place. Copy the following code from *https://invpy.com/median-template.py*

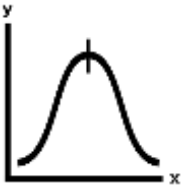
and paste it into your code editor. Replace the underscores with code to make a working program:

```
def median(numbers):
    # Special case: If the numbers list is empty, return None:
    if len(numbers) == ____:
        return ____

    # Sort the numbers list:
    ____.sort()

    # Get the index of the middle number:
    middleIndex = len(____) // ____

    # If the numbers list has an even length, return the average of the
    # middle two numbers:
    if len(numbers) % ____ == 0:
        return (numbers[____] + numbers[middleIndex - ____]) / ____
    # If the numbers list has an odd length, return the middlemost number:
    else:
        return numbers[____]
```

The complete solution for this exercise is given in Appendix A and *https://invpy.com/median.py*. You can view each step of this program as it runs under a debugger at *https://invpy.com/median-debug/*.

EXERCISE #16: MODE

```
mode([1, 1, 2, 3, 4])  →  1
```



Mode is the third statistical calculation exercise in this book. The mode is the number that appears most frequently in a list of numbers. Together with the median and average, you can get a descriptive summary of a list of numbers. This exercise tests your ability to use a dictionary to keep a count of the numbers in a list to find the most frequent number.

Exercise Description

Write a `mode()` function that has a `numbers` parameter. This function returns the mode, or most frequently appearing number, of the list of integer and floating-point numbers passed to the function.

These Python `assert` statements stop the program if their condition is `False`. Copy them to the bottom of your solution program. Your solution is correct if the following `assert` statements' conditions are all `True`:

```
assert mode([]) == None
assert mode([1, 2, 3, 4, 4]) == 4
assert mode([1, 1, 2, 3, 4]) == 1
import random
random.seed(42)
testData = [1, 2, 3, 4, 4]
for i in range(1000):
    random.shuffle(testData)
    assert mode(testData) == 4
```

Shuffling the order of the numbers should not affect the mode. The `for` loop does 1,000 such random shuffles to thoroughly check that this fact remains true. For an explanation of the `random.seed()` function, see the **Further Reading** section of Exercise #19, "Password Generator".

Try to write a solution based on the information in this description. If you still have trouble solving this exercise, read the **Solution Design and Special Cases and Gotchas** sections for additional hints.

Prerequisite concepts: `for` loops, augmented assignment operators, indexes, `not in` operator

Solution Design

The solution uses a dictionary to track how often each number appears in the list. The keys of the dictionary will be the number, and the values will be a count of how often the number appears in the list.

Start with an empty dictionary and set up two variables to keep track of the most frequent number and how many times this number has appeared in the list. Use a `for` loop to loop over numbers in the `numbers` list. If the current number we are looping on doesn't appear in the dictionary's keys, then create a key-value pair for it with the value starting at `0`. Then increment the count for this number in the dictionary. Finally, if this count is larger than the most frequent number's count, update the most frequent number variable and most frequent number's count variable with the current number and its count.

By the time the `for` loop has finished, the most frequent number variable contains the mode of the `numbers` list. Calculating the mode is similar Exercise #12, "Smallest & Biggest". Both solutions loop over the list of numbers, using another variable to (in Exercise #12) keep track of the smallest/biggest number found so far or (in this exercise) keep track of the most frequently occurring number found so far.

Special Cases and Gotchas

If the numbers parameter is an empty list, the function should return `None`. You should put the code that checks this at the start of the function.

Now try to write a solution based on the information in the previous sections. If you still have trouble solving this exercise, read the **Solution Template** section for additional hints.

Solution Template

Try to first write a solution from scratch. But if you have difficulty, you can use the following partial program as a starting place. Copy the following code from *https://invpy.com/mode-template.py* and paste it into your code editor. Replace the underscores with code to make a working program:

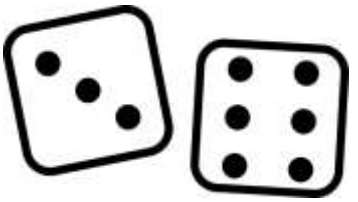
```
def mode(numbers):
    # Special case: If the numbers list is empty, return None:
    if len(numbers) == ____:
        return ____

    # Dictionary with keys of numbers and values of how often they appear:
    numberCount = {}

    # Track the most frequent number and how often it appears:
    mostFreqNumber = None
    mostFreqNumberCount = ____

    # Loop through all the numbers, counting how often they appear:
    for number in ____:
        # If the number hasn't appeared before, set it's count to 0.
        if ____ not in numberCount:
            numberCount[____] = ____
        # Increment the number's count:
        numberCount[____] += ____
        # If this is more frequent than the most frequent number, it
```

```
        # becomes the new most frequent number:
        if numberCount[number] > ____:
            mostFreqNumber = ____
            mostFreqNumberCount = ____[____]
    # The function returns the most frequent number:
    return mostFreqNumber
```

The complete solution for this exercise is given in Appendix A and *https://invpy.com/mode.py*. You can view each step of this program as it runs under a debugger at *https://invpy.com/mode-debug/*.

EXERCISE #17: DICE ROLL

```
rollDice(1)  →  6
```



This exercise uses Python's random number generating functions to simulate rolling any number of six-sided dice and returning the total sum of the dice roll. This exercise covers random number generation with Python's `random` module.

Exercise Description

Write a `rollDice()` function with a `numberOfDice` parameter that represents the number of six-sided dice. The function returns the sum of all of the dice rolls. For this exercise you must import Python's `random` module to call its `random.randint()` function for this exercise.

These Python `assert` statements stop the program if their condition is `False`. Copy them to the bottom of your solution program. Your solution is correct if the following `assert` statements' conditions are all `True`. We can't predict `rollDice()`'s random return value, but we can do repeated checks that the return value is within the correct range of expected values:

```
assert rollDice(0) == 0
assert rollDice(1000) != rollDice(1000)
for i in range(1000):
    assert 1 <= rollDice(1) <= 6
    assert 2 <= rollDice(2) <= 12
    assert 3 <= rollDice(3) <= 18
    assert 100 <= rollDice(100) <= 600
```

There is an astronomically small chance that the `assert rollDice(1000) != rollDice(1000)` will fail with a false positive. This assertion tests that `rollDice()` doesn't simply return the same "random" number each time by simulating rolling 1,000 dice twice and making sure the totals are different for both rolls. But, of course, there is a chance they'll come up the same total and cause an `AssertionError`. In that case, just rerun the assertion tests. After all, what are the odds of that happening twice?

Try to write a solution based on the information in this description. If you still have trouble solving this exercise, read the **Solution Design and Special Cases and Gotchas** sections for additional hints.

Prerequisite concepts: `import` statements, `random` module, `randint()`, `for` loops, `range()`, augmented assignment operator

Solution Design

First, create a variable to track the running total of the dice rolls and start it with the value `0`. Then make a `for` loop to loop the same number of times as the `numberOfDice` parameter. Inside the loop, call the `random.randint()` function to return a random number between `1` and `6` and add this number to the running total. After the loop completes, return the total.

If you don't know how the `random.randint()` function works, you can run `import random` and `help(random.randint)` in the interactive shell to view its documentation. Or you can do an internet search for "python randint" to find information about the function online.

Special Cases and Gotchas

There are no special cases for this exercise, but remember that Python's `random.randint()` function is inclusive of its arguments. Calling, for example, `range(6)` causes a `for` loop to loop from `0` up to, but not including, `6`. But calling `random.randint(1, 6)` returns a random number between `1` and `6`, including the `1` and the `6`. If you roll three 6-sided dice, the range is from 3 to 18, not 1 to 18.

Rolling multiple dice doesn't produce a uniform distribution: rolling two 6-sided dice is much more likely to come up with 7 because there are more combinations that add up to 7 (1 + 6, 2 + 5, 3 + 4, 4 + 3, 5 + 2, and 6 + 1) compared with rolling 2 (1 and 1 only). This is why you would need to call `random.randint(1, 6)` twice and add the rolls together instead of call `random.randint(2, 12)` once.

Your solution program needs `import random` at the top of the program in order to call the `random.randint()` function, or else you'll get a `NameError: name 'random' is not defined` error message.)

Now try to write a solution based on the information in the previous sections. If you still have trouble solving this exercise, read the **Solution Template** section for additional hints.

Solution Template

Try to first write a solution from scratch. But if you have difficulty, you can use the following partial program as a starting place. Copy the following code from *https://invpy.com/rolldice-template.py* and paste it into your code editor. Replace the underscores with code to make a working program:

```
# Import the random module for its randint() function.
____ random

def rollDice(numberOfDice):
    # Start the sum total at 0:
    total = ____
    # Run a loop for each die that needs to be rolled:
    for i in range(____):
        # Add the amount from one 6-sided dice roll to the total:
        total += random.randint(____, ____)
    # Return the dice roll total:
    return total
```

The complete solution for this exercise is given in Appendix A and *https://invpy.com/rolldice.py*. You can view each step of this program as it runs under a debugger at *https://invpy.com/rolldice-debug/*.

EXERCISE #18: BUY 8 GET 1 FREE

```
getCostOfCoffee(7, 2.50)  →  17.50
getCostOfCoffee(8, 2.50)  →  20
getCostOfCoffee(9, 2.50)  →  20
```

Let's say a coffee shop punches holes into a customer's card each time they buy a coffee. After the card has eight hole punches, the customer can use the card to get their 9th cup of coffee for free. In this exercise, you'll translate this into a simple calculation to see how much a given quantity of coffees costs while considering this buy-8-get-1-free system.

Exercise Description

Write a function named `getCostOfCoffee()` that has a parameters named `numberOfCoffees` and `pricePerCoffee`. Given this information, the function returns the total cost of the coffee order. This is not a simple multiplication of cost and quantity, however, because the coffee shop has an offer where you get one free coffee for every eight coffees you buy.

For example, buying eight coffees for $2.50 each costs $20 (or 8 × 2.5). But buying nine coffees also costs $20, since the first eight makes the ninth coffee free. Buying ten coffees calculates as follows: $20 for the first eight coffees, a free ninth coffee, and $2.50 for the tenth coffee for a total of $22.50.

These Python `assert` statements stop the program if their condition is `False`. Copy them to the bottom of your solution program. Your solution is correct if the following `assert` statements' conditions are all `True`:

```
assert getCostOfCoffee(7, 2.50) == 17.50
assert getCostOfCoffee(8, 2.50) == 20
assert getCostOfCoffee(9, 2.50) == 20
assert getCostOfCoffee(10, 2.50) == 22.50

for i in range(1, 4):
    assert getCostOfCoffee(0, i) == 0
    assert getCostOfCoffee(8, i) == 8 * i
    assert getCostOfCoffee(9, i) == 8 * i
    assert getCostOfCoffee(18, i) == 16 * i
    assert getCostOfCoffee(19, i) == 17 * i
    assert getCostOfCoffee(30, i) == 27 * i
```

Try to write a solution based on the information in this description. If you still have trouble solving this exercise, read the **Solution Design and Special Cases and Gotchas** sections for additional hints.

Prerequisite concepts: `while` loops, augmented assignment operator

Solution Design

I have two solutions for this exercise. The simpler one is a counting approach to this exercise. First, we create variables to track how much the total price is so far (this starts at `0`) and how many cups until we get a free cup of coffee (this starts at `8`). We then have a `while` loop that continues looping as long as there are still cups of coffee to count. Inside the loop, we decrement the `numberOfCoffees` argument and check if this is a free cup of coffee. If it is, we reset the number of cups to the next free cup back to 8. If it isn't a free coffee, we increase the total price and decrement the number of cups to the next free cup.

After the loop finishes, the function returns the variable tracking the total price.

Special Cases and Gotchas

There are several places where you can have an *off-by-one* error. These errors result from simple oversights that lead to slightly wrong calculations.

First, note that you must buy eight cups of coffee to get the ninth cup for free; you don't get the eighth cup for free. Also the free cup of coffee doesn't count towards the eight cups you buy to get a free cup. The price of eight cups and nine cups is the same: you can purposefully forego a free cup of coffee.

Now try to write a solution based on the information in the previous sections. If you still have trouble solving this exercise, read the **Solution Template** section for additional hints.

Solution Template

Try to first write a solution from scratch. But if you have difficulty, you can use the following partial program as a starting place. Copy the following code from *https://inventwithpython.com/buy8get1free-template.py* and paste it into your code editor. Replace the underscores with code to make a working program:

```
def getCostOfCoffee(numberOfCoffees, pricePerCoffee):
    # Track the total price:
    totalPrice = ____
    # Track how many coffees we have until we get a free one:
    cupsUntilFreeCoffee = 8

    # Loop until the number of coffees to buy reaches 0:
    while numberOfCoffees ____ 0:
        # Decrement the number of coffees left to buy:
        ____ -= 1

        # If this cup of coffee is free, reset the number to buy until
        # a free cup back to 8:
        if cupsUntilFreeCoffee == ____:
            ____ = 8
```

```
        # Otherwise, pay for a cup of coffee:
        else:
            # Increase the total price:
            totalPrice += ____
            # Decrement the coffees left until we get a free coffee:
            cupsUntilFreeCoffee -= ____

    # Return the total price:
    return ____
```

The complete solution for this exercise is given in Appendix A and *https://invpy.com/buy8get1free.py*. You can view each step of this program as it runs under a debugger at *https://invpy.com/buy8get1free-debug/*.

Another Solution Design

The counting solution design, while simple, gets slower the larger the number of purchased coffees becomes. If you called `getCostOfCoffee(1000000000, 2.50)` it could take a couple of minutes before the function returns an answer. There's a more direct way to calculate the total price for these large coffee orders.

First, you can calculate the number of free coffees by integer dividing `numberOfCoffees` by `9`. For every nine coffees, one is a free coffee. Any remainder coffees don't matter for counting the number of free coffees, which is why you use the `//` integer division operator instead of the `/` regular division operator. Store this division result in a variable named `numberOfFreeCoffees`.

To calculate the number of paid coffees, subtract `numberOfFreeCoffees` from `numberOfCoffees` and store this difference in a variable named `numberOfPaidCoffees`. (This makes sense; coffees are either paid for or free, so the number of paid and free coffees must add up to `numberOfCoffees`). The final value to return is the `numberOfPaidCoffees` times the `pricePerCoffee`.

The benefit of this solution design is that it does three calculations (a division, a subtraction, and a multiplication) no matter how big or small `numberOfCoffees`. Calling `getCostOfCoffee(1000000000, 2.50)` with this implementation finishes in milliseconds rather than minutes.

Another Solution Template

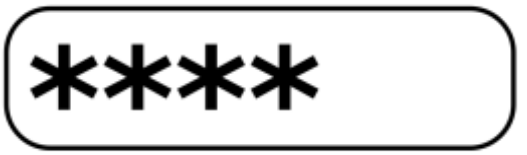
Try to first write a solution from scratch. But if you have difficulty, you can use the following partial program as a starting place. Copy the following code from *https://invpy.com/buy8get1free2-template.py* and paste it into your code editor. Replace the underscores with code to make a working program:

```
def getCostOfCoffee(numberOfCoffees, pricePerCoffee):
    # Calculate the number of free coffees we get in this order:
    numberOfFreeCoffees = ____ // 9

    # Calculate the number of coffees we will have to pay for in this order:
    numberOfPaidCoffees = numberOfCoffees - ____

    # Calculate and return the price:
    return ____ * ____
```

The complete solution for this exercise is given in Appendix A and *https://invpy.com/buy8get1free2.py*. You can view each step of this program as it runs under a debugger at *https://invpy.com/buy8get1free2-debug/*.

EXERCISE #19: PASSWORD GENERATOR

```
generatePassword(12)  →  'v*f6uoklQJ!d'
generatePassword(12)  →  ' Yzkr(j2T$MsG'
generatePassword(16)  →  ' UVp7ow8T%5LZl1la'
```



While a password made from a single English word like "rosebud" or "swordfish" is easy to remember, it isn't secure. A *dictionary attack* is when hackers program their computers to repeatedly try logging in with every word in the dictionary as the password. A dictionary attack won't work if you use randomly generated passwords. They may not be easy to remember, but they make hacking your accounts more difficult.

Exercise Description

Write a `generatePassword()` function that has a `length` parameter. The `length` parameter is an integer of how many characters the generated password should have. For security reasons, if `length` is less than `12`, the function forcibly sets it to 12 characters anyway. The password string returned by the function must have at least one lowercase letter, one uppercase letter, one number, and one special character. The special characters for this exercise are `~!@#$%^&*()_+`.

Your solution should import Python's `random` module to help randomly generate these passwords.

These Python `assert` statements stop the program if their condition is `False`. Copy them to the bottom of your solution program. Your solution is correct if the following `assert` statements' conditions are all `True`:

```
assert len(generatePassword(8)) == 12

pw = generatePassword(14)
assert len(pw) == 14
hasLowercase = False
hasUppercase = False
hasNumber = False
hasSpecial = False
for character in pw:
    if character in LOWER_LETTERS:
        hasLowercase = True
    if character in UPPER_LETTERS:
```

```
        hasUppercase = True
    if character in NUMBERS:
        hasNumber = True
    if character in SPECIAL:
        hasSpecial = True
assert hasLowercase and hasUppercase and hasNumber and hasSpecial
```

Try to write a solution based on the information in this description. If you still have trouble solving this exercise, read the **Solution Design and Special Cases and Gotchas** sections for additional hints.

Prerequisite concepts: `import` statements, `random` module, strings, string concatenation, `len()`, `append()`, `randint()`, `shuffle()`, `join()`

Solution Design

First, you'll need to create constant strings for each category of characters required by the exercise:

- Lowercase letters: abcdefghijklmnopqrstuvwxyz (26 characters)
- Uppercase letters: ABCDEFGHIJKLMNOPQRSTUVWXYZ (26 characters)
- Numbers: 1234567890 (10 characters)
- Special characters: ~!@#$%^&*()_+ (13 characters)

Next, create a string that concatenates all four strings into one 75-character string. These variables are *constants* in that they aren't meant to have their contents changed. By convention, constant variables are typed with ALL_UPPERCASE names and have underscores to separate words by convention. Constants are often created in the global scope outside of all functions, rather than as local variables inside a particular function. Constants are commonly used in all programming languages, even though the term "constant variable" is a bit of an oxymoron.

The first line of the `generatePassword()` function should check if the `length` argument is less than `12`, and if so, set `length` to `12`. Next, create a `password` variable that starts as an empty list. Then randomly select a character from the lowercase letter constant using Python's `random.randint()` function to pick a random integer index from the constant's string. Do this for the other three constants as well.

To guarantee that the final password has at least one character from each of the four categories, we'll begin the password with a character from each category. Then we'll keep adding characters from the combined string until the password reaches the required length.

But this isn't completely random since the first four characters are from predictable categories. To fix this issue, we'll call Python's `random.shuffle()` function to mix up the order of the characters. Unfortunately, the `random.shuffle()` function only works on lists, not strings, so we build up the password from an empty list rather than an empty string.

In a loop, keep adding a randomly selected character from the concatenated string with all characters until the `password` list is the same length as `length`. Then, pass the password list to `random.shuffle()` to mix up the order of the characters. Finally, combine this list of strings into a single string using `''.join(password)` and return it.

Special Cases and Gotchas

The `random.shuffle()` function only works with list values and not string values. This is why we add single-character strings to a list, shuffle it, and combine that list of strings into a single string with the `join()` string method. Otherwise, passing a string to `random.shuffle()` results in a `TypeError: 'str' object does not support item assignment` error message.

Now try to write a solution based on the information in the previous sections. If you still have trouble solving this exercise, read the **Solution Template** section for additional hints.

Solution Template

Try to first write a solution from scratch. But if you have difficulty, you can use the following partial program as a starting place. Copy the following code from *https://invpy.com/passwordgenerator-template.py* and paste it into your code editor. Replace the underscores with code to make a working program:

```
# Import the random module for its randint() function.
import ____

# Create string constants that for each type of character:
LOWER_LETTERS = 'abcdefghijklmnopqrstuvwxyz'
UPPER_LETTERS = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
NUMBERS = '1234567890'
SPECIAL = '~!@#$%^&*()_+'

# Create a string that has all of these strings combined:
ALL_CHARS = LOWER_LETTERS + ____ + ____ + ____

def generatePassword(length):
    # 12 is the minimum length for passwords:
    if length ____ 12:
        length = ____

    # Create a password variable that starts as an empty list:
    password = []
    # Add a random character from the lowercase, uppercase, digits, and
    # punctuation character strings:
    password.append(LOWER_LETTERS[random.randint(0, 25)])
    password.____(UPPER_LETTERS[random.randint(0, ____)])
    password.____(NUMBERS[random.randint(0, ____)])
    password.____(SPECIAL[random.randint(0, ____)])

    # Keep adding random characters from the combined string until the
    # password meets the length:
    while len(password) < ____:
        password.append(ALL_CHARS[random.randint(____, 74)])

    # Randomly shuffle the password list:
    random.shuffle(____)

    # Join all the strings in the password list into one string to return:
    return ''.join(____)
```

The complete solution for this exercise is given in Appendix A and *https://invpy.com/passwordgenerator.py*. You can view each step of this program as it runs under a

debugger at *https://inventwithpython.com/passwordgenerator-debug/*.

Further Reading

Most *random number generator* (RNG) algorithms are *pseudorandom*: they appear random but are actually predictable. Pseudorandom algorithms have an initial value (often an integer) called a seed, and the same starting seed value produces the same random numbers. For example, you can reset Python's seed by passing a seed integer to `random.seed()`. Notice how setting the same seed in the following interactive shell example produces the same sequence of "random" numbers:

```
>>> import random
>>> random.seed(42)  # Use any arbitrary integer for the seed.
>>> for i in range(20):
...   print(random.randint(0, 9), end=' ')
...
1 0 4 3 3 2 1 8 1 9 6 0 0 1 3 3 8 9 0 8
>>> random.seed(42)  # Reset using the same integer seed.
>>> for i in range(20):
...   print(random.randint(0, 9), end=' ')
...
1 0 4 3 3 2 1 8 1 9 6 0 0 1 3 3 8 9 0 8
```

Python can also generate truly random, not pseudorandom, numbers based on several sources of systems entropy such as boot time, process IDs, position of the mouse cursor, millisecond timing in between the last several keystrokes, and others. You can look up the official Python documentation for the `random.SystemRandom()` function at *https://docs.python.org/3/library/random.html*.

These theoretical hacks on pseudorandom numbers might concern you if you're an international spy or a journalist targeted by the intelligence agencies of nation-states, but in general, using this program is fine. Randomly- and pseudorandomly-generated passwords are still better than using a predictable password like "Stanley123". And many hacks happen because people have keystroke-reading malware on their computers or reuse the same password for multiple accounts. So if the password database of a website you use is hacked, those hackers can then try to identify the users' accounts on other popular websites and try those same passwords.

A *password manager* app is the most effective single thing a computer user can have to increase their security. When you need to log into a website, you enter your master password into the password manager app to unlock its encrypted database. Then you can copy the password from the manager to the clipboard. For example, your password could be something like GKfazu8WposcVP!EL8, but logging in with it is just a matter of pressing Ctrl-V to paste it into the website's login page.

I recommend the free and open-source KeePass app from *https://keepass.info*, but many password manager apps are freely available.

EXERCISE #20: LEAP YEAR

```
isLeapYear(2004)  →  True
```

It takes about 365.25 days for the earth to revolve around the sun. This slight offset would cause our 365-day calendar to become inaccurate over time. Therefore, leap years have an extra day, February 29th. A leap year occurs on all years divisible by four (e.g., 2016, 2020, 2024, and so on). However, the exception to this rule is that years divisible by one hundred (e.g., 2100, 2200, 2300, and so on) aren't leap years. And the exception to this exception is that years divisible by four hundred (e.g., 2000, 2400, and so on) are leap years.

Exercise Description

Write a `isLeapYear()` function with an integer `year` parameter. If `year` is a leap year, the function returns `True`. Otherwise, the function returns `False`.

These Python `assert` statements stop the program if their condition is `False`. Copy them to the bottom of your solution program. Your solution is correct if the following `assert` statements' conditions are all `True`:

```
assert isLeapYear(1999) == False
assert isLeapYear(2000) == True
assert isLeapYear(2001) == False
assert isLeapYear(2004) == True
assert isLeapYear(2100) == False
assert isLeapYear(2400) == True
```

Try to write a solution based on the information in this description. If you still have trouble solving this exercise, read the **Solution Design and Special Cases and Gotchas** sections for additional hints.

Prerequisite concepts: modulo operator, `elif` statements

Solution Design

Determining leap years involves checking if an integer is divisible by 4, 100, and 400. Solving Exercise #3, "Odd & Even," Exercise #5, "Fizz Buzz," and Exercise #6, "Ordinal Suffix," all involved the `%` modulo operator to determine the divisibility of integers. We'll use `year % 4`, `year % 100`, and `year % 400` in our solution.

It can also help to draw a flow chart of the general logic on a whiteboard or with paper and pen. It could look something like this:

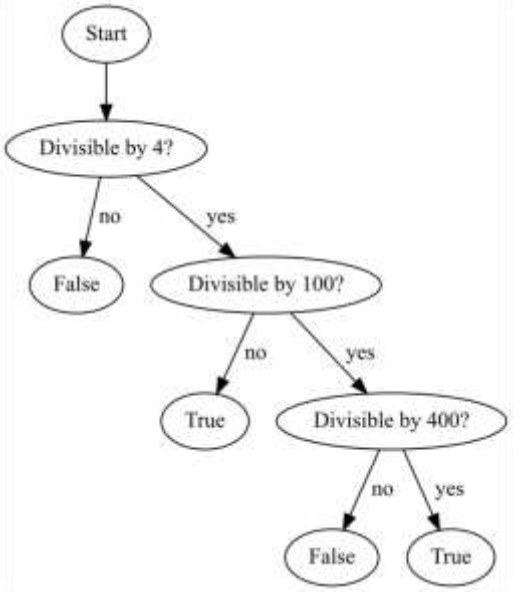


Figure 20-1: A flow chart showing the logical steps of determining if a year is a leap year.

Leap year rules have a few exceptions, so this function needs a set of `if`-`elif`-`else` statements. The order of these statements matters. You could use an `if` statement to check if the year is divisible by 4 and, if so, return `True`. But before returning, you need another `if` statement to check if the year is divisible by 100 and return `False`. But before that, you need yet another `if` statement that returns `True` if the year is divisible by 400. Writing code this way ends up looking like this:

```
def isLeapYear(year):
    if year % 4 == 0:
        if year % 100 == 0:
            if year % 400 == 0:
                # Year is divisible by 400:
                return True
            else:
                # Year is divisible by 100 but not by 400:
                return False
        else:
            # Year is divisible by 4 but not by 100:
            return True
    else:
        # Year is not divisible by 4:
        return False
```

This code works correctly but is hard to follow. There are several levels of indentation, and the nested `if`-`else` statements make it tricky to see which `else` statement is paired with which `if` statement.

Instead, try switching around the order of the logic. For example, if the year is divisible by 400, return `True`. Or else, if the year is divisible by 100, return `False`. Or else, if the year is divisible by 4, return `True`. Or else, for all other years return `False`. Writing code this way reduces the amount of nested `if`-`else` statements and produces more readable code.

Special Cases and Gotchas

Outside of the complex list of rules, there are no particular gotchas for this exercise. The tricky part is to ensure that you have the conditions for the `if` and `elif` statements in the correct order.

Now try to write a solution based on the information in the previous sections. If you still have trouble solving this exercise, read the **Solution Template** section for additional hints.

Solution Template

Try to first write a solution from scratch. But if you have difficulty, you can use the following partial program as a starting place. Copy the following code from *https://invpy.com/leapyear-template.py* and paste it into your code editor. Replace the underscores with code to make a working program:

```
def isLeapYear(year):
    # Years divisible by 400 are leap years:
    if ____ % 400 == ____:
        return ____
    # Otherwise, years divisible by 100 are not leap years:
    elif ____ % 100 == ____:
        return ____
    # Otherwise, years divisible by 4 are leap years:
    elif ____ % 4 == ____:
        return ____
    # Otherwise, every other year is not a leap year:
    else:
        return ____
```

The complete solution for this exercise is given in Appendix A and *https://invpy.com/leapyear.py*. You can view each step of this program as it runs under a debugger at *https://invpy.com/leapyear-debug/*.

EXERCISE #21: VALIDATE DATE

```
isValidDate(2005, 3, 29)  →  True
isValidDate(2005, 13, 32)  →  False
```

You can represent a date with three integers for the year, month, and day, but this doesn't mean that any integers represent a valid date. After all, there is no 13th month of the year or 32nd day of any month. This exercise has you check if a year/month/day combination is valid, given that different months have different numbers of days. You'll use the solution you wrote for Exercise #20, "Leap Year" as part of the solution for this exercise, so finish Exercise #20 before attempting this one.

Exercise Description

Write an `isValidDate()` function with parameters `year`, `month`, and `day`. The function should return `True` if the integers provided for these parameters represent a valid date. Otherwise, the function returns `False`. Months are represented by the integers `1` (for January) to `12` (for December) and days are represented by integers `1` up to `28`, `29`, `30`, or `31` depending on the month and year. Your solution should import your *leapyear.py* program from Exercise #20 for its `isLeapYear()` function, as February 29th is a valid date on leap years.

September, April, June, and November have 30 days. The rest have 31, except February which has 28 days. On leap years, February has 29 days.

These Python `assert` statements stop the program if their condition is `False`. Copy them to the bottom of your solution program. Your solution is correct if the following `assert` statements' conditions are all `True`:

```
assert isValidDate(1999, 12, 31) == True
assert isValidDate(2000, 2, 29) == True
assert isValidDate(2001, 2, 29) == False
assert isValidDate(2029, 13, 1) == False
assert isValidDate(1000000, 1, 1) == True
assert isValidDate(2015, 4, 31) == False
assert isValidDate(1970, 5, 99) == False
assert isValidDate(1981, 0, 3) == False
assert isValidDate(1666, 4, 0) == False

import datetime
d = datetime.date(1970, 1, 1)
oneDay = datetime.timedelta(days=1)
```

```
for i in range(1000000):
    assert isValidDate(d.year, d.month, d.day) == True
    d += oneDay
```

Try to write a solution based on the information in this description. If you still have trouble solving this exercise, read the **Solution Design and Special Cases and Gotchas** sections for additional hints.

Prerequisite concepts: `import` statements, Boolean operators, chaining operators, `elif` statements

Solution Design

Any integer is a valid value for the `year` parameter, but we do need to pass `year` to our `leapyear.isLeapYear()` function if `month` is set to `2` (February). Be sure your program has an `import leapyear` instruction and that the *leapyear.py* file is in the same folder as your solution's file.

Any `month` values outside of `1` to `12` would be an invalid date. In Python, you can chain together operators as a shortcut: the expression `1 <= month <= 12` is the same as the expression `(1 <= month) and (month <= 12)`. You can use this to determine if your `month` and `day` parameters are within a valid range.

The number of days in a month depends on the month:

- If `month` is `9`, `4`, `6`, or `11` (September, April, June, and November, respectively) the maximum value for `day` is `30`.
- If `month` is `2` (February), the maximum value for `day` is `28` (or `29` if `leapyear.isLeapYear(year)` returns `True`).
- Otherwise, the maximum value for day is `31`.

Like the solution for Exercise #20, "Leap Year," this solution is a series of `if`, `elif`, or `else` statements.

Special Cases and Gotchas

To simplify your code as much as possible, think about the cases that can cause the function to return as soon as possible. For example, if `month` is outside of the `1` to `12` range, the function returns `False`. There's no other set of circumstances you need to check; the function can immediately return `False`. If it doesn't return, you can assume that month contains a valid value for the rest of the function.

The other immediate check is if `leapyear.isLeapYear(year)` returns `True` and `month` is `2` and day is `29`, the function can immediately return `True`. If the function hasn't returned `True` for the leap day, you can assume that leap years are irrelevant for the rest of the code in the function.

Now try to write a solution based on the information in the previous sections. If you still have trouble solving this exercise, read the **Solution Template** section for additional hints.

Solution Template

Try to first write a solution from scratch. But if you have difficulty, you can use the following partial program as a starting place. Copy the following code from *https://inventwithpython.com/validatedate-*

template.py and paste it into your code editor. Replace the underscores with code to make a working program:

```
# Import the leapyear module for its isLeapYear() function:
____ leapyear

def isValidDate(year, month, day):
    # If month is outside the bounds of 1 to 12, return False:
    if not (1 ____ month ____ 12):
        return ____
    # After this point, you can assume the month is valid.

    # If the year is a leap year and the date is Feb 29th, it is valid:
    if leapyear.isLeapYear(____) and ____ == 2 and ____ == 29:
        return ____
    # After this point, you can assume the year is not a leap year.

    # Check for invalid dates in 31-day months:
    if month in (1, 3, 5, 7, 8, 10, 12) and not (1 <= ____ <= 31):
        return ____
    # Check for invalid dates in 30-day months:
    elif ____ in (4, 6, 9, 11) and not (1 <= day <= 30):
        return ____
    # Check for invalid dates in February:
    elif month == ____ and not (1 <= ____ <= 28):
        return ____

    # Date passes all checks and is valid, so return True:
    return ____
```

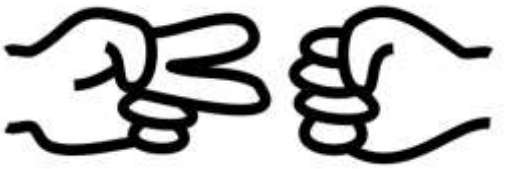
The complete solution for this exercise is given in Appendix A and *https://invpy.com/validatedate.py*. You can view each step of this program as it runs under a debugger at *https://invpy.com/validatedate-debug/*.

Further Reading

Python's `datetime` module has several features for dealing with dates and calendar data. You can learn more about it in Chapter 17 of *Automate the Boring Stuff with Python* at *https://automatetheboringstuff.com/2e/chapter17/*.

EXERCISE #22: ROCK, PAPER, SCISSORS

```
rpsWinner('rock', 'paper')  →  'player2'
```



Rock, paper, scissors is a popular hand game for two players. The two players simultaneously choose one of the three possible moves and determine the winner of the game: rock beats scissors, paper beats rock, and scissors beats paper. This exercise involves determining a game's outcome given the moves of the two players.

Exercise Description

Write a `rpsWinner()` function with parameters `player1` and `player2`. These parameters are passed one of the strings `'rock'`, `'paper'`, or `'scissors'` representing that player's move. If this results in player 1 winning, the function returns `'player one'`. If this results in player 2 winning, the function returns `'player two'`. Otherwise, the function returns `'tie'`.

These Python `assert` statements stop the program if their condition is `False`. Copy them to the bottom of your solution program. Your solution is correct if the following `assert` statements' conditions are all `True`:

```
assert rpsWinner('rock', 'paper') == 'player two'
assert rpsWinner('rock', 'scissors') == 'player one'
assert rpsWinner('paper', 'scissors') == 'player two'
assert rpsWinner('paper', 'rock') == 'player one'
assert rpsWinner('scissors', 'rock') == 'player two'
assert rpsWinner('scissors', 'paper') == 'player one'
assert rpsWinner('rock', 'rock') == 'tie'
assert rpsWinner('paper', 'paper') == 'tie'
assert rpsWinner('scissors', 'scissors') == 'tie'
```

Try to write a solution based on the information in this description. If you still have trouble solving this exercise, read the **Solution Design and Special Cases and Gotchas** sections for additional hints.

Prerequisite concepts: Boolean operators, `elif` statements

Solution Design

Similar to the solutions for Exercise #20, "Leap Year" and #21, "Validate Date", the solution for

this exercise is a set of `if`-`elif`-`else` statements. The `player1` parameter contains a string of the first player's move and the `player2` parameter contains a string of the second player's move. These strings will be one of `'rock'`, `'paper'`, and `'scissors'`. You'll want to use comparison operators to check the value of both players and join them with an `and` operator. For example, the expression `player1 == 'rock'` evaluates to `True` if the first player went with rock, and the expression `player2 == 'paper'` evaluates to `True` if the second player went with paper. This means that in the expression `player1 == 'rock' and player2 == 'paper'` evaluates to `True` if both sides of the `and` operator evaluated to `True`. In this case, the second player is the winner and the function should return `'player2'`.

Special Cases and Gotchas

You can check if the `player1` parameter equals the `player2` parameter at the start of the function and immediately return `'tie'` in that case.

Now try to write a solution based on the information in the previous sections. If you still have trouble solving this exercise, read the **Solution Template** section for additional hints.

Solution Template

Try to first write a solution from scratch. But if you have difficulty, you can use the following partial program as a starting place. Copy the following code from *https://invpy.com/rockpaperscissors-template.py* and paste it into your code editor. Replace the underscores with code to make a working program:

```
def rpsWinner(move1, move2):
    # Check all six possible combinations with a winner and return it:
    if move1 == 'rock' and move2 == 'paper':
        return 'player two'
    elif ____ == 'rock' and move2 == 'scissors':
        return ____
    ____ move1 == 'paper' and ____ == 'scissors':
        return ____
    ____ ____ == 'paper' and move2 == 'rock':
        return ____
    ____ move1 == 'scissors' and ____ == 'rock':
        return ____
    ____ ____ == 'scissors' and  ____ == 'paper':
        return 'player one'
    # For all other combinations, it is a tie:
    ____:
        return ____
```

The complete solution for this exercise is given in Appendix A and *https://invpy.com/rockpaperscissors.py*. You can view each step of this program as it runs under a debugger at *https://invpy.com/rockpaperscissors-debug/*.

EXERCISE #23: 99 BOTTLES OF BEER

```
99 bottles of beer on the wall,
      99 bottles of beer,
         Take one down,
        Pass it around,
98 bottles of beer on the wall,
```

"99 Bottles of Beer on the Wall" is a cumulative song often sung to pass the time (and annoy anyone close to the singer). Long, tedious activities are the perfect task for computers. In this exercise, you'll write a program to display the complete lyrics of this song.

Exercise Description

Write a program that displays the lyrics to "99 Bottles of Beer." Each stanza of the song goes like this:

X bottles of beer on the wall,

X bottles of beer,

Take one down,

Pass it around,

X – 1 bottles of beer on the wall,

The X in the song starts at 99 and decreases by one for each stanza. When X is one (and X – 1 is zero), the last line is "*No more bottles of beer on the wall!*" After each stanza, display a blank line to separate it from the next stanza.

You'll know you have the program correct if it matches the lyrics at *https://inventwithpython.com/bottlesofbeerlyrics.txt*. It looks like the following:

99 bottles of beer on the wall,

99 bottles of beer,

Take one down,

Pass it around,

98 bottles of beer on the wall,

98 bottles of beer on the wall,

98 bottles of beer,

Take one down,

Pass it around,

97 bottles of beer on the wall,

…cut for brevity…

1 bottle of beer on the wall,

1 bottle of beer,

Take one down,

Pass it around,

No more bottles of beer on the wall!

Try to write a solution based on the information in this description. If you still have trouble solving this exercise, read the **Solution Design and Special Cases and Gotchas** sections for additional hints.

Prerequisite concepts: `for` loops, `range()` with three arguments, `print()`

Solution Design

Use a `for` loop to loop from `99` down to, but not including, `1`. The 3-argument form of `range()` can do this with:

```
for numberOfBottles in range(99, 1, -1)
```

The `numberOfBottles` variable starts at the first argument, `99`. The second argument, `1`, is the value that `numberOfBottles` goes down to (but does not include). This means the last iteration sets `numberOfBottles` to `2`. The third argument, `-1`, is the *step argument* and changes `numberOfBottles` by `-1` instead of the default `1`. This causes the `for` loop to decrease the loop variable rather than increase it.

For example, if we run:

```
for i in range(4, 0, -1):
    print(i)
```

...it would produce the following output:

```
4
3
2
1
```

If we run:

```
for i in range(0, 8, 2):
    print(i)
```

...it would produce the following output:

```
0
2
4
6
```

The 3-argument form of `range()` allows you to set more detail than the 1-argument form, but the 1-argument form is more common because the start and step arguments are often the default `0` and `1`, respectively. For example, the code `for i in range(10)` is the equivalent of `for i in range(0, 10, 1)`.

Special Cases and Gotchas

Python's `print()` function accepts multiple arguments to display. These arguments can be strings, integers, Booleans, or values of any other data type. The `print()` function automatically converts the argument to a string. However, keep in mind that you can only concatenate a string value to another string value. So while these two lines of code are valid:

- `print(numberOfBottles, 'bottles of beer,')`
- `print(str(numberOfBottles) + ' bottles of beer,')`

This line of code produces a `TypeError: unsupported operand type(s) for +: 'int' and 'str'` error message because `numberOfBottles` holds an integer and the `+` operator cannot concatenate integers to strings.

- `print(numberOfBottles + ' bottles of beer,')`

The latter line of code is trying to pass a single argument to `print()`: the expression `numberOfBottles + ' bottles of beer,'` evaluates to this single value. This differs from the first two examples where two arguments, separated by a comma, are passed to `print()`. The difference is a subtle but important. While `print()` automatically converts arguments to strings, string concatenation doesn't and requires you to call `str()` to perform this conversion.

Even the word "convert" is a bit of a misnomer here: function calls return brand new values rather than change an existing value. Calling `len('hello')` *returns* the integer value `5`. It's not that the string `'hello'` has been *changed* to the integer `5`. Similarly, calling `str(42)` or `int('99')` doesn't change the argument but returns new values.

This is why code such as `str(numberOfBottles)` doesn't convert the integer in the `numberOfBottles` variable to a string. Instead, if you want to change the variable, you need to assign the return value to the variable like:

```
numberOfBottles = str(numberOfBottles)
```

Don't think of this as modifying the value in `numberOfBottles`, but rather replacing it with the value returned from the `str()` function call.

Now try to write a solution based on the information in the previous sections. If you still have trouble solving this exercise, read the **Solution Template** section for additional hints.

Solution Template

Try to first write a solution from scratch. But if you have difficulty, you can use the following partial program as a starting place. Copy the following code from *https://invpy.com/bottlesofbeer-*

template.py and paste it into your code editor. Replace the underscores with code to make a working program:

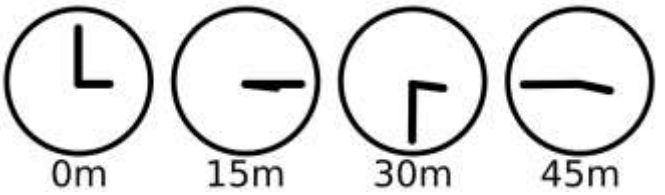
```
# Loop from 99 to 2, displaying the lyrics to each stanza.
for numberOfBottles in range(99, 1, ____):
    print(____, 'bottles of beer on the wall,')
    print(____, 'bottles of beer,')
    ____('Take one down,')
    ____('Pass it around,')

    # If there is only one, print "bottle" instead of "bottles".
    if (numberOfBottles - 1) == ____:
        ____('1 bottle of beer on the wall,')
    ____:
        print(____ - 1, ' bottles of beer on the wall,')

# The last stanza has singular "bottle" and a different final line:
____('1 bottle of beer on the wall,')
____('1 bottle of beer,')
____('Take one down,')
____('Pass it around,')
____('No more bottles of beer on the wall!')
```

The complete solution for this exercise is given in Appendix A and *https://invpy.com/bottlesofbeer.py*. You can view each step of this program as it runs under a debugger at *https://invpy.com/bottlesofbeer-debug/*.

Further Reading

Project #50 in my book, *The Big Book of Small Python Projects*, also implements this "99 bottles of beer" exercise. You can read it for free online at *https://inventwithpython.com/bigbookpython/*. Project #51 is a version where the lyrics deteriorate over time with erased letters, swapped letters, and other drunken typos.

EXERCISE #24: EVERY 15 MINUTES



Clocks have an unusual counting system compared to the normal decimal number system we're familiar with. Instead of beginning at 0 and going to 1, 2, and so on forever, clocks start at 12 and go on to 1, 2, and so on up to 11. Then it loops back to 12 again. (Clocks are quite odd if you think about it: 12 am comes before 11 am and 12 pm comes before 11 pm.) This is a bit more complicated than simply writing a program that counts upward. This exercise requires using nested `for` loops to loop over the minutes, the hours, and the am and pm halve of the day.

Exercise Description

Write a program that displays the time for every 15 minute interval from 12:00 am to 11:45 pm. Your solution should produce the following output:

```
12:00 am
12:15 am
12:30 am
12:45 am
1:00 am
1:15 am
--cut--
11:30 pm
11:45 pm
```

There are 96 lines in the full output.

Try to write a solution based on the information in this description. If you still have trouble solving this exercise, read the **Solution Design and Special Cases and Gotchas** sections for additional hints.

Prerequisite concepts: `for` loops, lists, nested loops, string concatenation

Solution Design

This solution requires the use of three nested `for` loops. The outermost `for` loop iterates over `'am'` and `'pm'`. The second `for` loop iterates over twelve hours, starting with `'12'`, then `'1'`, then `'2'`, and so on until `'11'`. The third `for` loop iterates over the minutes in 15-minute increments: `'00'`, `'15'`, `'30'`, and `'45'`. Note that the hours and minutes values are strings, not integers, because we need to concatenate them into our final string, like: `'12' + ':' + '00' + ' ' + 'am'` evaluates to `'12:00 am'`

You're used to `for` loops iterating over a range of integers from the `range()` function. But

Python's `for` loops can iterate over lists of any values. For example, enter the following into the interactive shell:

```
>>> for i in ['Alice', 'Bob', 'Carol']:
...     print('Hello ' + i)
...
Hello Alice
Hello Bob
Hello Carol
```

What a `for` loop does is iterate over a sequence of values. The following interactive shell example is the equivalent `for i in range(4)`:

```
>>> for i in [0, 1, 2, 3]:
...     print(i)
...
0
1
2
3
```

In this case, we explicitly typed out the integers to iterate in a list rather than use the more convenient `range(4)`. But they produce identical results. And explicitly typing out the integers in a list becomes prohibitively long for large ranges such as `range(1000)`.

Special Cases and Gotchas

The order of the nested `for` loops is important. You want the innermost `for` loop to iterate over minutes, the next innermost to iterate over hours, and the outermost `for` loop to iterate over `'am'` and `'pm'`.

Now try to write a solution based on the information in the previous sections. If you still have trouble solving this exercise, read the **Solution Template** section for additional hints.

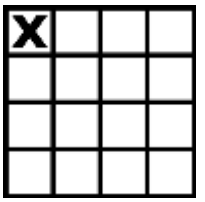
Solution Template

Try to first write a solution from scratch. But if you have difficulty, you can use the following partial program as a starting place. Copy the following code from *https://invpy.com/every15minutes-template.py* and paste it into your code editor. Replace the underscores with code to make a working program:

```
# Loop over am and pm:
for meridiem in [____, 'pm']:
    # Loop over every hour:
    for hour in [____, '1', '2', '3', '4', '5', '6', '7', '8', '9', '10', '11']:
        # Loop over every 15 minutes:
        for minutes in ['00', ____, ____, '45']:
            # Print the time:
            print(____ + ':' + ____ + ' ' + ____)
```

The complete solution for this exercise is given in Appendix A and *https://invpy.com/every15minutes.py*. You can view each step of this program as it runs under a debugger at *https://invpy.com/every15minutes-debug/*.

EXERCISE #25: MULTIPLICATION TABLE



Learning the multiplication table is an early part of our childhood math education. The multiplication table shows every product of two single digit numbers. In this exercise, we print a multiplication table on the screen using nested `for` loops and some string manipulation to align the columns correctly.

Exercise Description

Write a program that displays a multiplication table that looks like this:

```
  | 1  2  3  4  5  6  7  8  9 10
--+------------------------------
 1| 1  2  3  4  5  6  7  8  9 10
 2| 2  4  6  8 10 12 14 16 18 20
 3| 3  6  9 12 15 18 21 24 27 30
 4| 4  8 12 16 20 24 28 32 36 40
 5| 5 10 15 20 25 30 35 40 45 50
 6| 6 12 18 24 30 36 42 48 54 60
 7| 7 14 21 28 35 42 49 56 63 70
 8| 8 16 24 32 40 48 56 64 72 80
 9| 9 18 27 36 45 54 63 72 81 90
10|10 20 30 40 50 60 70 80 90 100
```

The number labels along the top and left sides are the numbers to multiply, and where their column and row intersect is the product of those numbers. Notice that the single-digit numbers are padded with spaces to keep them aligned in the same column. You may use Python's `rjust()` string method to provide this padding. This method returns a string with space characters added on the left side to right-justify the text, and the **Solution Design** section explains how it works.

The line along the top side of the table is made up of minus sign characters. The line along the left side is made up of vertical pipe characters (above the Enter key on the keyboard). A plus sign marks their intersection. Your solution is correct if the output matches the above text of the multiplication table. You can use a simple `print()` call for the number labels and lines at the top of the table. However, don't *hard code* the text of the multiplication table into your program: your program should be more than just a bunch of `print()` calls.

Try to write a solution based on the information in this description. If you still have trouble solving this exercise, read the **Solution Design and Special Cases and Gotchas** sections for additional hints.

Prerequisite concepts: `print()`, `for` loops, `range()` with two arguments, `end` keyword argument for `print()`, `rjust()`, `str()`

Solution Design

In the first part of the program, print out the horizontal number labels and separating line. You can program these two lines directly with two `print()` calls:

```
print('  | 1  2  3  4  5  6  7  8  9 10')
print('--+---------------------------')
```

Remember that you need the appropriate amount of spaces in between the numbers so the columns of the multiplication table to line up. You can treat all numbers as though they were two digits. The single-digit numbers should have a space printed on their left side, making the string right-justified. Python's `rjust()` string method can do this for you. Enter the following into the interactive shell:

```
>>> '42'.rjust(4)  # Adds two spaces.
'  42'
>>> '042'.rjust(4)  # Adds one space.
' 042'
>>> '0042'.rjust(4)  # Adds zero spaces.
'0042'
```

Notice how all of the strings returned from the `rjust(4)` call are four characters long. If the original string is less than four characters long, the `rjust()` method puts spaces on the left side of the returned string until it is four characters long.

To print out the multiplication table, two nested `for` loops can iterate over each product. The outermost `for` loop iterates over the numbers of each row, and the innermost `for` loop iterates over the numbers of each column in the current row. You don't want a newline to appear after each product, but only after each row of products. Python's `print()` function automatically adds a newline to the end of the string you pass. To disable this, pass a blank string for the `end` keyword argument like `print('Some text', end='')`.

A simplified version of this code would look like this:

```
>>> for row in range(1, 11):
...     for column in range(1, 11):
...         print(str(row * column) + ' ', end='')
...     print()  # Print a newline.
...
1 2 3 4 5 6 7 8 9 10
2 4 6 8 10 12 14 16 18 20
3 6 9 12 15 18 21 24 27 30
4 8 12 16 20 24 28 32 36 40
5 10 15 20 25 30 35 40 45 50
6 12 18 24 30 36 42 48 54 60
7 14 21 28 35 42 49 56 63 70
8 16 24 32 40 48 56 64 72 80
9 18 27 36 45 54 63 72 81 90
10 20 30 40 50 60 70 80 90 100
```

Your solution needs these products appropriately aligned as well as the number labels along the top and left side.

Special Cases and Gotchas

The easiest mistake is getting the amount of padding wrong, causing the columns to no longer be aligned. For example, if you fail to pad the single-digit products with an extra space character, the

multiplication table ends up looking like the misaligned table in the previous section.

Now try to write a solution based on the information in the previous sections. If you still have trouble solving this exercise, read the **Solution Template** section for additional hints.

Solution Template

Try to first write a solution from scratch. But if you have difficulty, you can use the following partial program as a starting place. Copy the following code from *https://invpy.com/multiplicationtable-template.py* and paste it into your code editor. Replace the underscores with code to make a working program:

```
# Print the heading of each column:
print('  | 1  2  3  4  5  6  7  8  9 10')
print('--+------------------------------')

# Loop over all numbers from 1 to 10:
for column in range(____, ____):
    # Print the number label on the right side:
    print(str(____).rjust(____) + '|', end=____)

    # Loop over all numbers from 1 to 10:
    for row in range(____, ____):
        # Print the product, padded to two digits, followed by a space:
        print(str(____).rjust(____) + ' ', end=____)
    # After the loop, print a newline to end the row:
    ____()
```

The complete solution for this exercise is given in Appendix A and *https://invpy.com/multiplicationtable.py*. You can view each step of this program as it runs under a debugger at *https://invpy.com/multiplicationtable-debug/*.

Further Reading

A similar multiplication table project is also used in the free book, *The Big Book of Small Python Projects*, as Project #49 at *https://inventwithpython.com/bigbookpython*. If you are interested in chemistry, that book also has a project that displays the periodic table of elements.

EXERCISE #26: HANDSHAKES

There is only one handshake that can happen between two people. Between three people, there are three possible handshaking pairs. Between four people, there are six handshakes; five people, ten handshakes, and so on. This exercise explores the full range of possible handshaking combinations with nested `for` loops.

Exercise Description

Write a function named `printHandshakes()` with a list parameter named `people` which will be a list of strings of people's names. The function prints out `'X shakes hands with Y'`, where X and Y are every possible pair of handshakes between the people in the list. No duplicates are permitted: if "Alice shakes hands with Bob" appears in the output, then "Bob shakes hands with Alice" should not appear.

For example, `printHandshakes(['Alice', 'Bob', 'Carol', 'David'])` should print:

```
Alice shakes hands with Bob
Alice shakes hands with Carol
Alice shakes hands with David
Bob shakes hands with Carol
Bob shakes hands with David
Carol shakes hands with David
```

The `printHandshakes()` function must also return an integer of the number of handshakes.

These Python `assert` statements stop the program if their condition is `False`. Copy them to the bottom of your solution program. Your solution is correct if the output displays all possible handshakes and the following `assert` statements' conditions are all `True`:

```
assert printHandshakes(['Alice', 'Bob']) == 1
assert printHandshakes(['Alice', 'Bob', 'Carol']) == 3
assert printHandshakes(['Alice', 'Bob', 'Carol', 'David']) == 6
```

Try to write a solution based on the information in this description. If you still have trouble solving this exercise, read the **Solution Design and Special Cases and Gotchas** sections for additional hints.

Prerequisite concepts: `for` loops, `range()` with two arguments, `len()`, augmented assignment operators

Solution Design

We need a pair of nested `for` loops to obtain the pairs of people in each handshake. The outer `for` loop iterates over each index in the person list for the first handshaker, and the inner `for` loop iterates over each index in the person list after the outer loop's index.

The pattern behind the movements of `i` and `j` are easier to see when visually laid out, as in Figure 26-1, which uses a 5-item `people` list as an example. The indexes `i` and `j` refer to the two people in the handshake:

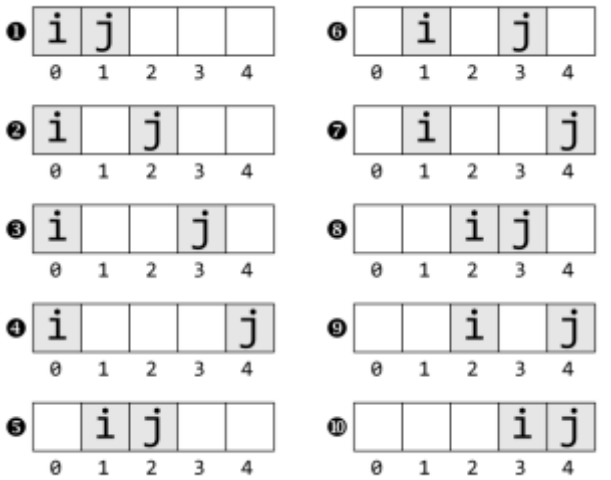


Figure 26-1: The pattern of i and j's movement.

As the algorithm runs, `j` starts after `i` and moves to the right, and when it reaches the end, `i` moves right once and `j` starts after `i` again. In the above example with 5 people (indexes `0` to `4`) `i` starts at `0` and `j` starts at `i + 1`, or `1`. The `j` variable increments until it reaches `4`, at which point `i` increments to `1` and `j` resets back to `i + 1`, which is now `2`.

If you look at the overall range of `i` and `j`, you'll see that `i` starts at index `0` and ends at the second to last index. Meanwhile, `j` starts at the index after `i` and ends at the last index. This means our nested `for` loops over the `people` list parameter would look like this:

```
for i in range(0, len(people) - 1):
    for j in range(i, len(people)):
```

This solution is identical to the nested `for` loops in Exercise #42, "Bubble Sort."

Special Cases and Gotchas

The most common mistake you want to avoid is having repeated handshakes. This can happen if your nested `for` loops cover the full range of indexes in the people list like so:

```
for i in range(0, len(people)):
    for j in range(0, len(people)):
```

In this case, `i` and `j` would run with each pair twice: for example, the first time with `people[i]` as the first handshaker and `people[j]` as the second handshaker, and then with `people[i]` as the second handshaker and `people[j]` as the first handshaker.

Now try to write a solution based on the information in the previous sections. If you still have trouble solving this exercise, read the **Solution Template** section for additional hints.

Solution Template

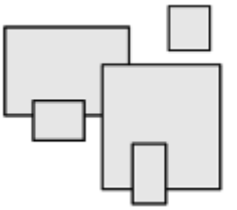
Try to first write a solution from scratch. But if you have difficulty, you can use the following partial program as a starting place. Copy the following code from *https://inventwithpython.com/handshake-template.py* and paste it into your code editor. Replace the underscores with code to make a working program:

```
def printHandshakes(people):
    # The total number of handshakes starts at 0:
    numberOfHandshakes = ____
    # Loop over every index in the people list except the last:
    for i in range(0, len(____) - 1):
        # Loop over every index in the people list after index i:
        for j in range(i + ____, len(____)):
            # Print a handshake between the people at index i and j:
            print(people[____], 'shakes hands with', people[____])
            # Increment the total number of handshakes:
            numberOfHandshakes += ____
    # Return the total number of handshakes:
    return numberOfHandshakes
```

The complete solution for this exercise is given in Appendix A and *https://inventwithpython.com/handshake.py*. You can view each step of this program as it runs under a debugger at *https://inventwithpython.com/handshake-debug/*.

EXERCISE #27: RECTANGLE DRAWING

```
drawRectangle(16, 4)  →  ################
                         ################
                         ################
                         ################
```



In this exercise, you'll create some *ASCII art*, primitive graphics created from text characters. There will be a few such exercises in this book. In this first one, your code draws a solid rectangle out of `#` hashtag characters.

Exercise Description

Write a `drawRectangle()` function with two integer parameters: `width` and `height`. The function doesn't return any values but rather prints a rectangle with the given number of hashtags in the horizontal and vertical directions.

There are no Python `assert` statements to check the correctness of your program. Instead, you can visually inspect the output yourself. For example, calling `drawRectangle(10, 4)` should produce the following output:

```
##########
##########
##########
##########
```

If either the `width` or `height` parameter is `0` or a negative number, the function should print nothing.

Try to write a solution based on the information in this description. If you still have trouble solving this exercise, read the **Solution Design and Special Cases and Gotchas** sections for additional hints.

Prerequisite concepts: `for` loops, `range()`, `print()`, `end` keyword argument for `print()`

Solution Design

The solution requires a pair of nested `for` loops. The inner `for` loop prints a row of hashtag

characters the width of the `width` parameter, while the outer `for` loop prints a number of rows the same as the `height` parameter. Inside the inner loop, prevent `print()` from automatically printing a newline by passing the `end=''` keyword argument, like `print('#', end='')`.

Alternatively, you can use string replication to create a row of hashtag characters. In Python, you can use the `*` operator with a string and an integer to evaluate to a longer string. For example, enter the following into the interactive shell:

```
>>> 'Hello' * 3
'HelloHelloHello'
>>> '#' * 16
'################'
>>> width = 10
>>> width * '#'
'##########'
```

Using string replication, you can avoid needing a second `for` loop in your solution.

Special Cases and Gotchas

The solution to this exercise is fairly straightforward. But you should note that in terminal windows, the text characters are twice as tall as they are wide. If you want to display a square shape, the width you pass to the `drawRectangle()` function should be twice the height.

Now try to write a solution based on the information in the previous sections. If you still have trouble solving this exercise, read the **Solution Template** section for additional hints.

Solution Template

Try to first write a solution from scratch. But if you have difficulty, you can use the following partial program as a starting place. Copy the following code from *https://invpy.com/rectangledrawing-template.py* and paste it into your code editor. Replace the underscores with code to make a working program:

```
def drawRectangle(width, height):
    # Special case: If width or height is less than 1, draw nothing:
    if width ____ 1 or height ____ 1:
        return

    # Loop over each row:
    for row in range(____):
        # Loop over each column in this row:
        for column in range(width):
            # Print a hashtag:
            print('#', ____='')
        # At the end of the row, print a newline:
        print()
```

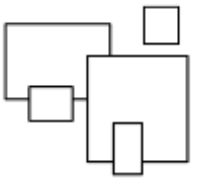
The complete solution for this exercise is given in Appendix A and *https://invpy.com/rectangledrawing.py*. You can view each step of this program as it runs under a debugger at *https://invpy.com/rectangledrawing-debug/*.

Further Reading

For examples of ASCII art, check out *https://en.wikipedia.org/wiki/ASCII_art* and *https://www.asciiart.eu/*. I've also compiled a large number of ASCII art examples in the *.txt* text files in this Git repo: *https://github.com/asweigart/asciiartjsondb*

EXERCISE #28: BORDER DRAWING

```
drawBorder(16, 4)  ➔  +--------------+
                      |              |
                      |              |
                      +--------------+
```



Similar to the solid, filled-in ASCII art rectangles our code generated in Exercise #27, "Rectangle Drawing," this exercise draws only the border of a rectangle. The `+` plus character is used for the corners, the `-` dash character for horizontal lines, and the `|` pipe character for vertical lines. (This is the similar style as the lines in Exercise #25's multiplication table.

Exercise Description

Write a `drawBorder()` function with parameters `width` and `height`. The function draws the border of a rectangle with the given integer sizes. There are no Python `assert` statements to check the correctness of your program. Instead, you can visually inspect the output yourself. For example, calling `drawBorder(16, 4)` would output the following:

```
+--------------+
|              |
|              |
+--------------+
```

The interior of the rectangle requires printing spaces. The sizes given include the space required for the corners. If the `width` or `height` parameter is less than `2`, the function should print nothing.

Try to write a solution based on the information in this description. If you still have trouble solving this exercise, read the **Solution Design and Special Cases and Gotchas** sections for additional hints.

Prerequisite concepts: Boolean operators, strings, string concatenation, string replication, `for` loops, `range()`

Solution Design

There are three separate parts required for the `drawBorder()` function: drawing the top border line, drawing the middle, and drawing the bottom border line. The code for drawing the top and bottom border line will be identical. So really, there's only two parts you need to code in this function.

Drawing the top horizontal line involves creating a string with a `+` plus character on the left, followed by a number of `-` minus characters, and then another `+` plus character on the right. The number of `-` minus characters needed is `width - 2`, because the two `+` plus characters for the corners count as two units of width.

Similarly, drawing the middle rows requires a `|` pipe character, followed by a number of space characters, and then another `|` pipe character. The number of spaces is also `width - 2`. You'll also need to put this code in a `for` loop, and draw a number of these rows equal to `height - 2`.

Finally, drawing the bottom horizontal line is identical to drawing the top. You can copy and paste the code.

String replication can easily create the `-` minus and space character strings. In Python, you can use the `*` operator with a string and an integer to evaluate to a longer string. For example, enter the following into the interactive shell:

```
>>> 'Hello' * 3
'HelloHelloHello'
>>> '-' * 16
'----------------'
>>> width = 10
>>> (width - 2) * '-'
'--------'
```

Special Cases and Gotchas

Note that the minimum width and height for a border is `2`. Calling `drawBorder(2, 2)` should print the following on the screen:

```
++
++
```

If either the `width` or `height` argument is less than `2`, the function prints nothing.

Now try to write a solution based on the information in the previous sections. If you still have trouble solving this exercise, read the **Solution Template** section for additional hints.

Solution Template

Try to first write a solution from scratch. But if you have difficulty, you can use the following partial program as a starting place. Copy the following code from *https://invpy.com/borderdrawing-template.py* and paste it into your code editor. Replace the underscores with code to make a working program:

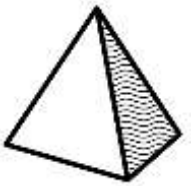
```
def drawBorder(width, height):
    # Special case: If the width or height is less than two, draw nothing:
    if width < ____ or height < ____:
        return

    # Print the top row:
```

```
    print('+' + ('-' * (width - ____)) + ____)

    # Loop for each row (except the top and bottom):
    for i in range(height - 2):
        # Print the sides:
        print(____ + (____ * (width - 2)) + ____)

    # Print the bottom row:
    print(_____________________________________)
```

The complete solution for this exercise is given in Appendix A and *https://invpy.com/borderdrawing.py*. You can view each step of this program as it runs under a debugger at *https://invpy.com/borderdrawing-debug/*.

EXERCISE #29: PYRAMID DRAWING

```
drawPyramid(5)  ➔       #
                       ###
                      #####
                     #######
                    #########
```



This exercise continues the generative ASCII art programs of Exercise #27, "Rectangle Drawing," and Exercise #28, "Border Drawing." In this exercise, your code prints a pyramid of hashtag characters in any given size.

Exercise Description

Write a `drawPyramid()` function with a `height` parameter. The top of the pyramid has one centered hashtag character, and the subsequent rows have two more hashtags than the previous row. The number of rows matches the `height` integer. There are no Python `assert` statements to check the correctness of your program. Instead, you can visually inspect the output yourself. For example, calling `drawPyramid(8)` would output the following:

```
       #
      ###
     #####
    #######
   #########
  ###########
 #############
###############
```

Try to write a solution based on the information in this description. If you still have trouble solving this exercise, read the **Solution Design and Special Cases and Gotchas** sections for additional hints.

Prerequisite concepts: strings, string concatenation, string replication, `for` loops, `range()`

Solution Design

It's important to notice the general pattern as the height of the pyramid increases. Here's a pyramid of height 1:

```
#
```

Here's a pyramid of height 2:

```
 #
###
```

Here's a pyramid of height 3:

```
  #
 ###
#####
```

In order to center the pyramid correctly, you need to print the correct amount of space characters on the left side of the row. Here's the pyramid with `height` set to `5` and the spaces marked with periods to make them visible:

```
....#
...###
..#####
.#######
#########
```

The number of hashtag characters begins at `1` for the top row and then increases by `2`. The code in the `for` loop requires a number of hashtags equal to `rowNumber * 2 + 1`. The following table shows the number of spaces and hashtags for a pyramid of height 5:

Row Number	Number of Spaces	Number of Hashtags
0	4	1
1	3	3
2	2	5
3	1	7
4	0	9

Notice that for all pyramids, the number of spaces for the top row is `height - 1`, and each subsequent row has one less space than the previous row. An easier way to create each row is with a `for` loop that ranges from `0` up to, but not including, `height` for the row number, where row 0 is at the top. Then the number of spaces at a row number is `height - (rowNumber + 1)`.

String replication can easily create the `#` hashtag and space character strings. In Python, you can use the `*` operator with a string and an integer to evaluate to a longer string. For example, enter the following into the interactive shell:

```
>>> 'Hello' * 3
'HelloHelloHello'
>>> '#' * 7
'#######'
```

```
>>> rowNumber = 5
>>> (rowNumber * 2 + 1) * '#'
'###########'
```

Special Cases and Gotchas

If our solution uses a `for` loop to loop over the range from `0` up to, but not including, `height`, we don't need a separate check for a `height` of `0` or a negative `height`. This is because these values cause the `for` loop to not run its code, resulting in no output.

Now try to write a solution based on the information in the previous sections. If you still have trouble solving this exercise, read the **Solution Template** section for additional hints.

Solution Template

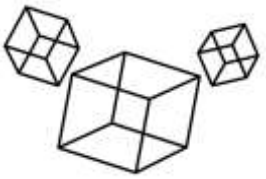
Try to first write a solution from scratch. But if you have difficulty, you can use the following partial program as a starting place. Copy the following code from *https://invpy.com/pyramiddrawing-template.py* and paste it into your code editor. Replace the underscores with code to make a working program:

```
def drawPyramid(height):
    # Loop over each row from 0 up to height:
    for rowNumber in range(____):
        # Create a string of spaces for the left side of the pyramid:
        leftSideSpaces = ' ' * (____ - (rowNumber + ____))
        # Create the string of hashtags for this row of the pyramid:
        pyramidRow = '#' * (____ * 2 + ____)
        # Print the left side spaces and the row of the pyramid:
        ____(leftSideSpaces + pyramidRow)
```

The complete solution for this exercise is given in Appendix A and *https://invpy.com/pyramiddrawing.py*. You can view each step of this program as it runs under a debugger at *https://invpy.com/pyramiddrawing-debug/*.

EXERCISE #30: 3D BOX DRAWING

```
drawBox(2)  ➔     +-----+
                 /     /|
                /     / |
               +-----+  +
               |     | /
               |     |/
               +-----+
```



In this exercise, we'll move from 2D ASCII art into 3D ASCII art by programmatically generating boxes at any given size.

Exercise Description

Write a `drawBox()` function with a `size` parameter. The `size` parameter contains an integer for the width, length, and height of the box. The horizontal lines are drawn with `-` dash characters, the vertical lines with `|` pipe characters, and the diagonal lines with `/` forward slash characters. The corners of the box are drawn with `+` plus signs.

There are no Python `assert` statements to check the correctness of your program. Instead, you can visually inspect the output yourself. For example, calling `drawBox(1)` through `drawBox(5)` would output the following boxes, respectively:

```
                                                    +----------+
                                                   /          /|
                                  +--------+      /          / |
                                 /        /|     /          /  |
                      +------+  /        / |    /          /   |
                     /      /| /        /  |   /          /    |
          +----+    /      / | /        /   |  +----------+     +
         /    /|   /      /  | +--------+    +  |          |    /
  +--+  /    / |  +------+   + |        |   /   |          |   /
 /  /| +----+  +  |      |  /  |        |  /    |          |  /
+--+ + |    | /   |      | /   |        | /     |          | /
|  |/  |    |/    |      |/    |        |/      |          |/
+--+   +----+     +------+     +--------+       +----------+

Size 1  Size 2      Size 3         Size 4            Size 5
```

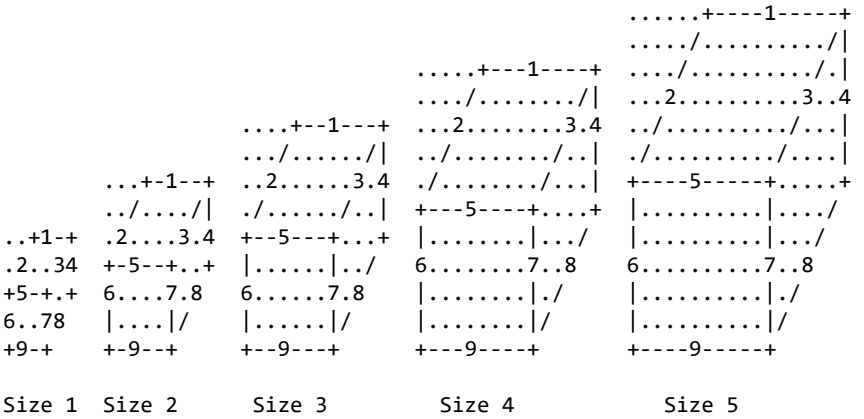
If the argument for `size` is less than `1`, the function prints nothing.

Try to write a solution based on the information in this description. If you still have trouble solving this exercise, read the **Solution Design and Special Cases and Gotchas** sections for additional hints.

Prerequisite concepts: strings, string concatenation, string replication, `for` loops, `range()`

Solution Design

This exercise is a significant leap in complexity compared to the previous rectangle, border, and pyramid drawing exercises. Nine different lines must be drawn, as well as several whitespace areas on the left side and interior of the box. However, solving this exercise is still a matter of figuring out the sizing patterns. Drawing the boxes manually in a text editor first can help you determine the pattern behind the boxes' lines. Here are the boxes from size 1 to 5 with the lines numbered and the whitespace marked with periods (since spaces' invisibility makes them hard to count):

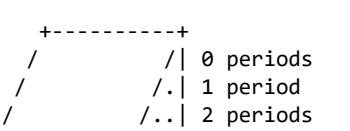


Because `print()` calls display text left-to-right and top-to-bottom, we'll have to consider the lines and whitespace in that order. For the following descriptions, note that `size` is the integer parameter passed to the `drawBox()` function.

The box's diagonal lines follow the pattern of having `size` slash characters. The box's vertical lines follow the pattern of having `size` pipe characters. Meanwhile, the horizontal lines made of `size * 2` dash characters. Look at the largest box on the right of the above diagram: The horizontal lines 1, 5, and 9 are made of 10 `-` dash characters (that is, `size * 2`). The diagonal lines 2, 3, and 8 are made of 5 `/` slash characters (that is, `size`). The vertical lines 4, 6, and 7 are also made of 5 `|` pipe characters (that is, `size`).

The horizontal lines 1, 5, and 9 are identical: They're made of a `+` plus character, followed by `size * 2` dash characters and another `+` plus character. Line 1 has a number of spaces to the left of the line that equals `size + 1`.

The interior spaces for the top and front surfaces of the box are `size` space characters, the same as the number of `-` dash characters. The interior space of the right surface of the box is trickier. For example, here's a box with size as 5 with the right-side surface spaces marked with periods:



```
  /          /...| 3 periods
 /          /....| 4 periods
+----------+.....+ 5 periods
|          |..../  4 periods
|          |.../   3 periods
|          |../    2 periods
|          |./     1 period
|          |/      0 periods
+----------+

   Size 5
```

As you print the top surface, the right-side surface has an increasing number of spaces ranging from `0` to `size - 1` before printing the `|` pipe character of line 4. When you print line 5, the right-side surface has exactly `size` spaces before printing the `+` plus sign for the corner. And as you print the front surface, the right-side surface has a decreasing number of spaces ranging from `size - 1` to `0` before printing the `/` slash character of line 8.

Finally, you'll print the `+` plus and `-` dash characters of line 9 at the bottom.

Special Cases and Gotchas

There's nothing unexpected about printing these boxes. While there are many things to keep track of, always remember that the number of space and dash characters you print in each row is always relative to the `size` parameter. There will be `size` pipe and slash characters in each vertical line, and `size * 2` dash characters in each horizontal line.

Now try to write a solution based on the information in the previous sections. If you still have trouble solving this exercise, read the **Solution Template** section for additional hints.

Solution Template

Try to first write a solution from scratch. But if you have difficulty, you can use the following partial program as a starting place. Copy the following code from *https://inventwithpython.com/boxdrawing-template.py* and paste it into your code editor. Replace the underscores with code to make a working program:

```
def drawBox(size):
    # Special case: Draw nothing if size is less than 1:
    if size < ____:
        return

    # Draw back line on top surface:
    print(' ' * (____ + 1) + '+' + '-' * (____ * 2) + '+')

    # Draw top surface:
    for i in range(____):
        print(' ' * (____ - i) + '/' + ' ' * (____ * 2) + '/' + ' ' * i + '|')

    # Draw top line on top surface:
    print(____ + ____ * (size * 2) + ____ + ' ' * size + '+')

    # Draw front surface:
    for i in range(size - 1, ____, ____):
        print(____ + ' ' * (size * ____) + ____ + ' ' * i + ____)
```

```
    # Draw bottom lie on front surface:
    print(____ + ____ * (size * 2) + ____)

# In a loop, call drawBox() with arguments 1 to 5:
for i in range(1, 6):
    drawBox(i)
```

The complete solution for this exercise is given in Appendix A and *https://inventwithpython.com/boxdrawing.py*. You can view each step of this program as it runs under a debugger at *https://inventwithpython.com/boxdrawing-debug/*.

Further Reading

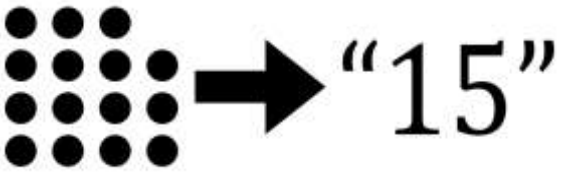
If you enjoy the challenge of these generative ASCII art exercises, check out *https://github.com/asweigart/programmedpatterns/*. This website has several growing patterns that you can try to replicate as Python programs. For example, one such pattern looks like this:

```
#     #       #       #
##    ##      ##      ##
      ###     ###     ###
              ####    ####
                      #####
```

The first four steps of the pattern are provided. You could then write a function with a `step` parameter that prints the pattern at that given step. There are hundreds of patterns featured on the site.

EXERCISE #31: CONVERT INTEGERS TO STRINGS

```
convertIntToStr(42)  →  '42'
```



In Python, the values `42` and `'42'` are two different values: you can perform mathematics on the integer `42`, but the string `'42'` is the same as any other two-character text string. You can perform mathematical addition on integer values, but not on strings. And you can concatenate or replicate string values, but not integers. A common programming task is to obtain the string equivalent of a number. You can use the `str()` function to do this conversion but in this exercise you'll recreate this function yourself.

Exercise Description

Write a `convertIntToStr()` function with an `integerNum` parameter. This function operates similarly to the `str()` function in that it returns a string form of the parameter. For example, `convertIntToStr(42)` should return the string `'42'`. The function doesn't have to work for floating-point numbers with a decimal point, but it should work for negative integer values.

Avoid using Python's `str()` function in your code, as that would do the conversion for you and defeat the purpose of this exercise. However, we use `str()` with `assert` statements to check that your `convertIntToStr()` function works the same as `str()` for all integers from `-10000` to `9999`:

```
for i in range(-10000, 10000):
    assert convertIntToStr(i) == str(i)
```

Try to write a solution based on the information in this description. If you still have trouble solving this exercise, read the **Solution Design and Special Cases and Gotchas** sections for additional hints.

Prerequisite concepts: dictionaries, `while` loops, string concatenation, integer division

Solution Design

To create the string equivalent of the integer value, we'll convert the integers to strings one digit at a time. Let's create a variable named `stringNum` to hold the converted string. In a loop, the expression `integerNum % 10` evaluates to the digit in the one's place of `integerNum`. Store this

result in a variable named `onesPlaceDigit`.

Then we add the string equivalent of `onesPlaceDigit` to `stringNum` with a series of `if-elif` statements such as the following:

```
if onesPlaceDigit == 0:
    stringNum = '0' + stringNum
elif onesPlaceDigit == 1:
    stringNum = '1' + stringNum
elif onesPlaceDigit == 2:
    stringNum = '2' + stringNum
elif onesPlaceDigit == 3:
    stringNum = '3' + stringNum
elif onesPlaceDigit == 4:
    stringNum = '4' + stringNum
elif onesPlaceDigit == 5:
    stringNum = '5' + stringNum
elif onesPlaceDigit == 6:
    stringNum = '6' + stringNum
elif onesPlaceDigit == 7:
    stringNum = '7' + stringNum
elif onesPlaceDigit == 8:
    stringNum = '8' + stringNum
elif onesPlaceDigit == 9:
    stringNum = '9' + stringNum
```

However, this code is pretty long and tedious. A more concise and common approach is to create a dictionary that maps individual integer digits to their string equivalents, and then look up the value for the `onesPlaceDigit` key:

```
DIGITS_INT_TO_STR = {0: '0', 1: '1', 2: '2', 3: '3', 4: '4', 5: '5', 6: '6', 7: '7',
8: '8', 9: '9'}
stringNum = DIGITS_INT_TO_STR[onesPlaceDigit] + stringNum
```

After this, we can integer divide it by `10` to "remove" the digit in the one's place of `integerNum`. For example, `41096 // 10` evaluates to `4109`, effectively removing the `6` from the number and making the `9` the new digit in the one's place to convert. Our loop can continue looping and converting digits until `integerNum` is `0`. For example, doing this to the integer 41096 would carry out the following operations:

- 41096 // 10 = 4109
- 4109 // 10 = 410
- 410 // 10 = 41
- 41 // 10 = 4
- 4 // 10 = 0

At this point the algorithm is finished and `stringNum` contains the string form.

Special Cases and Gotchas

At the start of the function, check if the `integerNum` parameter is `0` and, if so, immediately return the string `'0'`. Our algorithm is such that otherwise it will return the blank string `''` for `0`, which is incorrect.

Now try to write a solution based on the information in the previous sections. If you still have trouble solving this exercise, read the **Solution Template** section for additional hints.

Solution Template

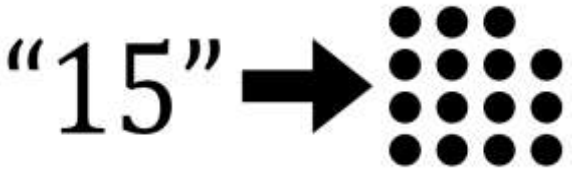
Try to first write a solution from scratch. But if you have difficulty, you can use the following partial program as a starting place. Copy the following code from *https://invpy.com/convertinttostr-template.py* and paste it into your code editor. Replace the underscores with code to make a working program:

```
def convertIntToStr(integerNum):
    # Special case: Check if integerNum is 0, and return '0' if so:
    if integerNum == ____:
        return ____

    # This dictionary maps single integer digits to string digits:
    DIGITS_INT_TO_STR = {0: '0', 1: '1', 2: '2', 3: '3', 4: '4',
        5: '5', 6: '6', 7: '7', 8: '8', 9: '9'}

    # Make a note whether the number is negative or not, and make
    # integerNum positive for the rest of the function's code:
    if integerNum < ____:
        isNegative = ____
        integerNum = ____
    else:
        isNegative = ____

    # stringNum holds the converted string, and starts off blank:
    stringNum = ____

    # Keeping looping while integerNum is greater than zero:
    while integerNum ____ 0:
        # Mod the integerNum by 10 to get the digit in the ones place:
        onesPlaceDigit = integerNum % ____
        # Put the corresponding string digit at the front of stringNum:
        stringNum = DIGITS_INT_TO_STR[onesPlaceDigit] + ____
        # Divide integerNum by ten to remove one entire digit place:
        integerNum //= ____

    # If the number was originally negative, add a minus sign:
    if isNegative:
        return ____ + stringNum
    else:
        return ____
```

The complete solution for this exercise is given in Appendix A and *https://invpy.com/convertinttostr.py*. You can view each step of this program as it runs under a debugger at *https://invpy.com/convertinttostr-debug/*.

EXERCISE #32: CONVERT STRINGS TO INTEGERS

```
convertStrToInt('42')  →  42
```



To complement Exercise #31, “Convert Integers to Strings”, in this exercise we’ll convert strings of numeric digits into their integer equivalents. The most common use case for this is taking the string returned from, say, the `input()` function or a text file’s `read()` method and converting it to an integer to perform mathematical operations on it. You can use Python’s `int()` function to do this conversion, but in this exercise, you’ll recreate this function yourself.

Exercise Description

Write a `convertStrToInt()` function with a `stringNum` parameter. This function returns an integer form of the parameter just like the `int()` function. For example, `convertStrToInt('42')` should return the integer `42`. The function doesn’t have to work for floating-point numbers with a decimal point, but it should work for negative number values.

Avoid using `int()`in your code, as that would do the conversion for you and defeat the purpose of this exercise. However, we do use `int()` with `assert` statements to check that your `convertStrToInt()` function works the same as `int()` for all integers from `-10000` to `9999`:

```
for i in range(-10000, 10000):
    assert convertStrToInt(str(i)) == i
```

Try to write a solution based on the information in this description. If you still have trouble solving this exercise, read the **Solution Design and Special Cases and Gotchas** sections for additional hints.

Solution Design

The solution for this exercise is quite different than the int-to-string algorithm. Still, they are both similar in that they convert one digit and use a dictionary to map between string digits and integer digits:

```
DIGITS_STR_TO_INT = {'0': 0, '1': 1, '2': 2, '3': 3, '4': 4, '5': 5, '6': 6, '7': 7,
'8': 8, '9': 9}
```

The function creates an `integerNum` variable to hold the integer form of `stringNum` as we build it. This variable starts with the value `0`. Your code must also note if there is a minus sign at the start of the string, in which case

Our algorithm loops over the individual digits in the `stringNum` parameter, starting on the left and moving right. The code multiplies current integer in `integerNum` by 10 to "move" all of these digits to the left by one place, then adds the current digit.

For example, if we needed to convert the string `'41096'` to an integer, the code needs to carry out the following operations:

- `integerNum` = `0`
- `integerNum` = `(0 * 10) + 4` = `4`
- `integerNum` = `(4 * 10) + 1` = `41`
- `integerNum` = `(41 * 10) + 0` = `410`
- `integerNum` = `(410 * 10) + 9` = `4109`
- `integerNum` = `(4109 * 10) + 6` = `41096`

Before returning, we convert this integer to a negative number if the original string began with a minus sign.

Special Cases and Gotchas

The `convertStrToInt()` function must be able to handle strings representing negative integers. To do this, check if `stringNum[0]` (the first character in the string) is the `'-'` dash character. If so, we can mark an `isNegative` variable to `True` (and `False` otherwise). Then we can remove this dash character by setting `stringNum = stringNum[1:]`, replacing the string in `stringNum` with a string of all the characters in `stringNum` after the first.

At the end of the function, the function can return `-integerNum` if `isNegative` was set to `True`.

Now try to write a solution based on the information in the previous sections. If you still have trouble solving this exercise, read the **Solution Template** section for additional hints.

Solution Template

Try to first write a solution from scratch. But if you have difficulty, you can use the following partial program as a starting place. Copy the following code from *https://inventwithpython.com/convertstrtoint-template.py* and paste it into your code editor. Replace the underscores with code to make a working program:

```
def convertStrToInt(stringNum):
    # This dictionary maps string digits to single integer digits:
    DIGITS_STR_TO_INT = {'0': 0, '1': 1, '2': 2, '3': 3, '4': 4,
        '5': 5, '6': 6, '7': 7, '8': 8, '9': 9}

    # Make a note whether the number is negative or not, and make
    # integerNum positive for the rest of the function's code:
    if stringNum[0] == '-':
        isNegative = True
```

```
        stringNum = stringNum[1:]  # Remove the negative sign.
    else:
        isNegative = False

    # integerNum holds the converted integer, and starts off at 0:
    integerNum = 0

    # Loop over the digits in the string from left to right:
    for i in range(len(stringNum)):
        # Get the integer digit from the string digit:
        digit = DIGITS_STR_TO_INT[stringNum[i]]
        # Add this to the integer number:
        integerNum = (integerNum * 10) + digit

    # If the number was originally negative, make the integer
    # negative before returning it:
    if isNegative:
        return -integerNum
    else:
        return integerNum
```

The complete solution for this exercise is given in Appendix A and *https://invpy.com/convertstrtoint.py*. You can view each step of this program as it runs under a debugger at *https://invpy.com/convertstrtoint-debug/*.

EXERCISE #33: COMMA-FORMATTED NUMBERS

```
commaFormat(12345)  →  '12,345'
```

In the US and UK, the digits of numbers are grouped with commas every three digits. For example, the number 79033516 is written as 79,033,516 for readability. In this exercise, you'll write a function that takes a number and returns a string of the number with comma formatting.

Exercise Description

Write a `commaFormat()` function with a `number` parameter. The argument for this parameter can be an integer or floating-point number. Your function returns a string of this number with proper US/UK comma formatting. There is a comma after every third digit in the whole number part. There are no commas at all in the fractional part: The proper comma formatting of 1234.5678 is 1,234.5678 and not 1,234.567,8.

These Python `assert` statements stop the program if their condition is `False`. Copy them to the bottom of your solution program. Your solution is correct if the following `assert` statements' conditions are all `True`:

```
assert commaFormat(1) == '1'
assert commaFormat(10) == '10'
assert commaFormat(100) == '100'
assert commaFormat(1000) == '1,000'
assert commaFormat(10000) == '10,000'
assert commaFormat(100000) == '100,000'
assert commaFormat(1000000) == '1,000,000'
assert commaFormat(1234567890) == '1,234,567,890'
assert commaFormat(1000.123456) == '1,000.123456'
```

Try to write a solution based on the information in this description. If you still have trouble solving this exercise, read the **Solution Design and Special Cases and Gotchas** sections for additional hints.

Prerequisite concepts: strings, `str()`, `in` operator, `index()`, slices, string concatenation

Solution Design

Despite involving numbers, this exercise is actually about text manipulation. The characters of the string just happen to be numeric digits.

First, we convert the `number` argument to a string with the `str()` function. This will work whether the number is an integer or a floating-point number. Once we have the `number` as a string, we can check for the existence of a period which indicates it was a floating-point number with a fractional part. The expression `'.' in number` evaluates to `True` if the string in number has a period character. Next, we can use `number.index('.')` to find the index of this period character. (The `index()` method raises a `ValueError` exception if `'.'` doesn't appear in the string, but the previous `'.'` in number expression being `True` guarantees that it does.)

We need to remove this fractional part from `number` while saving it in another variable to add back in later. This way we are only adding commas to the whole number part of the `number` argument, whether or not it was an integer or floating-point number.

Next, let's start variables named `triplet` and `commaNumber` as blank strings. As we loop over the digits of `number`, the `triplet` variable will store digits until it has three of them, at which point we add them to `commaNumber` (which contains the comma-formatted version of number) with a comma. The first time we add `triplet` to `commaNumber`, there will be an extra comma at the end of a number. For example, the triplet `'248'` gets added to `commaNumber` as `'248,'`. We can remove the extra comma just before returning the number.

We need to loop starting at the one's place in the number and moving left, so our `for` loop should work in reverse: `for i in range(len(number) - 1, -1, -1)`. For example, if `number` is `4096`, then the first iteration of the loop can access `number[3]`, the second iteration can access `number[2]`, and so on. This way the first triplet ends up being `'096'` instead of `'409'`.

If the loop finishes and there are leftover digits in `triplet`, add them to `commaNumber` with a comma. Finally, return `commaNumber` except with the comma at the end truncated: `commaNumber[:-1]` evaluates to everything in `commaNumber` except the last character.

Finally, we need to add the fractional part back in the number if there was one originally.

Special Cases and Gotchas

Several bugs that can occur in our code. We should consider them ahead of writing our code so we can ensure they don't sneak past us. These bugs could include:

- A comma at the end of number, e.g., `386` producing `'386,'`
- A comma at the front of a number, e.g., `499000` producing `',499,000'`
- Commas appearing in the fraction part, e.g., `12.3333` producing `'12.3,333'`
- Grouping triplets in reverse order, e.g., `4096` producing `'409,6'`

However you tackle this exercise, ensure that your code doesn't make any of these mistakes.

Now try to write a solution based on the information in the previous sections. If you still have trouble solving this exercise, read the **Solution Template** section for additional hints.

Solution Template

Try to first write a solution from scratch. But if you have difficulty, you can use the following partial program as a starting place. Copy the following code from *https://invpy.com/commaformat-template.py* and paste it into your code editor. Replace the underscores with code to make a working program:

```
def commaFormat(number):
    # Convert the number to a string:
    number = str(____)

    # Remember the fractional part and remove it from the number, if any:
    if '.' in ____:
        fractionalPart = number[number.index(____):]
        number = number[:number.index('.')]
    else:
        fractionalPart = ''

    # Create a variable to hold triplets of digits and the
    # comma-formatted string as it is built:
    triplet = ____
    commaNumber = ____

    # Loop over the digits starting on the right side and going left:
    for i in range(len(number) - 1, ____, ____):
        # Add the digits to the triplet variable:
        triplet = ____[i] + ____
        # When the triplet variable has three digits, add it with a
        # comma to the comma-formatted string:
        if ____(triplet) == ____:
            commaNumber = triplet + ',' + ____
            # Reset the triplet variable back to a blank string:
            triplet = ____

    # If the triplet has any digits left over, add it with a comma
    # to the comma-formatted string:
    if triplet != '':
        commaNumber = ____ + ',' + ____

    # Return the comma-formatted string:
    return ____[:____] + fractionalPart
```

The complete solution for this exercise is given in Appendix A and *https://invpy.com/commaformat.py*. You can view each step of this program as it runs under a debugger at *https://invpy.com/commaformat-debug/*.

EXERCISE #34: UPPERCASE LETTERS

```
getUppercase('Hello')  →  'HELLO'
```

Python is known as a "batteries included" language because its standard library comes with many useful functions and modules. One of these is the `upper()` string method, which returns an uppercase version of the string: `'Hello'.upper()` evaluates to `'HELLO'`. However, in this exercise, you'll create your own implementation of this method.

Exercise Description

Write a `getUppercase()` function with a `text` string parameter. The function returns a string with all lowercase letters in `text` converted to uppercase. Any non-letter characters in `text` remain as they are. For example, `'Hello'` causes `getUppercase()` to return `'HELLO'` but `'goodbye 123!'` returns `'GOODBYE 123!'`.

These Python `assert` statements stop the program if their condition is `False`. Copy them to the bottom of your solution program. Your solution is correct if the following `assert` statements' conditions are all `True`:

```
assert getUppercase('Hello') == 'HELLO'
assert getUppercase('hello') == 'HELLO'
assert getUppercase('HELLO') == 'HELLO'
assert getUppercase('Hello, world!') == 'HELLO, WORLD!'
assert getUppercase('goodbye 123!') == 'GOODBYE 123!'
assert getUppercase('12345') == '12345'
assert getUppercase('') == ''
```

Try to write a solution based on the information in this description. If you still have trouble solving this exercise, read the **Solution Design and Special Cases and Gotchas** sections for additional hints.

Prerequisite concepts: `for` loops, `in` operator, string concatenation, indexes

Solution Design

The `getUppercase()` function should start with a new, empty string that will only contain non-lowercase characters. Then, we can use a loop to go over each character in the `text` parameter, copying characters to this new string. If the character is a lowercase letter, we can copy the uppercase

version of that letter. Otherwise, a non-lowercase letter character can be copied to the new string as-is. After the loop finishes, `getUppercase()` returns the newly-built uppercase string.

Getting the uppercase version of a letter will involve a dictionary that maps lowercase letters to uppercase letters. If a character from the `text` parameter exists as a key in the dictionary, we know it is a letter and the dictionary contains its corresponding uppercase version. This uppercase letter is concatenated to the end of the returned string. Otherwise, the original character from `text` is concatenated to the returned string.

Special Cases and Gotchas

The `getUppercase()` function should work equally well whether the `text` parameter is in lowercase or already in uppercase. Also, any non-letter characters aren't affected by the `getUppercase()` function.

Note that using the dictionary that maps lowercase letters to uppercase letters means our program only works for the basic 26 letters of the English alphabet. Therefore, it can't convert letters with accent marks to uppercase, such as `'ñ'` to `'Ñ'`, the way Python's `upper()` string method can. You would have to add every accented letter to the dictionary if you want it to be converted to uppercase.

Now try to write a solution based on the information in the previous sections. If you still have trouble solving this exercise, read the **Solution Template** section for additional hints.

Solution Template

Try to first write a solution from scratch. But if you have difficulty, you can use the following partial program as a starting place. Copy the following code from *https://inventwithpython.com/uppercase-template.py* and paste it into your code editor. Replace the underscores with code to make a working program:

```
# Map the lowercase letters to uppercase letters.
LOWER_TO_UPPER = {'a': 'A', 'b': 'B', 'c': 'C', 'd': 'D', 'e': 'E', 'f': 'F', 'g':
'G', 'h': 'H', 'i': 'I', 'j': 'J', 'k': 'K', 'l': 'L', 'm': 'M', 'n': 'N', 'o': 'O',
'p': 'P', 'q': 'Q', 'r': 'R', 's': 'S', 't': 'T', 'u': 'U', 'v': 'V', 'w': 'W', 'x':
'X', 'y': 'Y', 'z': 'Z'}

def getUppercase(text):
    # Create a new variable that starts as a blank string and will
    # hold the uppercase form of text:
    uppercaseText = ''
    # Loop over all the characters in text, adding non-lowercase
    # characters to our new string:
    for character in ____:
        if character in ____:
            # Append the uppercase form to the new string:
            uppercaseText += ____[____]
        else:
            uppercaseText += ____

    # Return the uppercase string:
    return ____
```

The complete solution for this exercise is given in Appendix A and *https://inventwithpython.com/uppercase.py*. You can view each step of this program as it runs under a debugger at *https://inventwithpython.com/uppercase-*

debug/.

EXERCISE #35: TITLE CASE

`getTitleCase('cat dog moose')` → `'Cat Dog Moose'`

In this exercise, you'll have to convert a string to title case where every word in the string begins with an uppercase letter. The remaining letters in the word are in lowercase. Title case is a slight increase in complexity compared to Exercise #34, "Uppercase Letters", so I advise that you solve that exercise before attempting this one.

Exercise Description

Write a `getTitleCase()` function with a `text` parameter. The function should return the title case form of the string: every word begins with an uppercase and the remaining letters are lowercase. Non-letter characters separate words in the string. This means that `'Hello World'` is considered to be two words while `'HelloWorld'` is considered to be one word. Not only spaces, but all non-letter characters can separate words, so `'Hello5World'` and `'Hello@World'` also have two words.

Python's `upper()` and `lower()` string methods return uppercase and lowercase forms of the string, and you can use these in your implementation. You may also use the `isalpha()` string method, which returns `True` if the string contains only uppercase or lowercase letter characters. However, you may not use Python's `title()` string method, as that would defeat the purpose of the exercise. Similarly, while you need to split up a string into individual words, don't use Python's `split()` string method.

These Python `assert` statements stop the program if their condition is `False`. Copy them to the bottom of your solution program. Your solution is correct if the following `assert` statements' conditions are all `True`:

```
assert getTitleCase('Hello, world!') == 'Hello, World!'
assert getTitleCase('HELLO') == 'Hello'
assert getTitleCase('hello') == 'Hello'
assert getTitleCase('hElLo') == 'Hello'
assert getTitleCase('') == ''
assert getTitleCase('abc123xyz') == 'Abc123Xyz'
assert getTitleCase('cat dog RAT') == 'Cat Dog Rat'
assert getTitleCase('cat,dog,RAT') == 'Cat,Dog,Rat'

import random
random.seed(42)
chars = list('abcdefghijklmnopqrstuvwxyz1234567890 ,.')
```

```
for i in range(1000):
    random.shuffle(chars)
    assert getTitleCase(''.join(chars)) == ''.join(chars).title()
```

The code in the `for` loop generates random strings and checks that your `getTitleCase()` function returns the same string that Python's built-in `title()` string method does. This allows us to quickly generate 1,000 test cases for your solution.

Try to write a solution based on the information in this description. If you still have trouble solving this exercise, read the **Solution Design and Special Cases and Gotchas** sections for additional hints.

Prerequisite concepts: strings, `for` loops, `range()`, `len()`, `upper()`, `isalpha()`, `lower()`

Solution Design

The main challenge in this exercise isn't converting letters to uppercase and lowercase but splitting the string up into individual words. We don't need to use Python's `split()` string method or the advanced regular expressions library. Look at the three example strings with the first letter of each word highlighted in Figure 35-1.

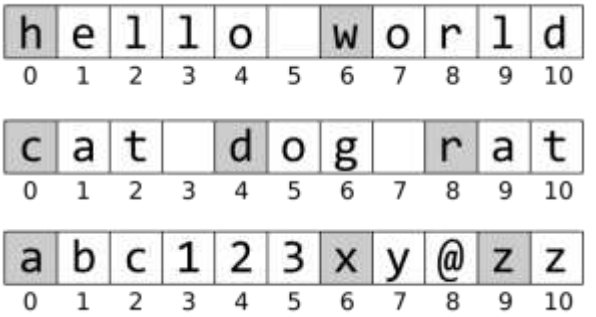


Figure 35-1: Three strings with the first letter of every word highlighted.

By looking at these examples, we can figure out that what makes a character in the string the first letter of a word is that the character is either the first character of the string (at index `0`) or follows a non-letter character. Our title case string will have these letters in uppercase and every other letter lowercase. Non-letter characters remain as they are.

Our function can start with a variable named `titledText` that holds the title case string form of the `text` parameter as we build it. Then a `for` loop can loop over all the indexes of the string. If the index is `0` (meaning it is at the start of the string) or the character at the previous index is not a letter, add the uppercase form of the character to `titledText`. Otherwise, add the lowercase form of the character to `titledText`.

Note that Python's `upper()` and `lower()` string methods have no effect on strings of non-letter characters. The expression `'42!'.upper()` and `'42!'.lower()` both evaluate to `'42!'`.

By the time the `for` loop has finished, `titledText` contains the complete title case form of text for the function to return.

Special Cases and Gotchas

Title case not only means the first letter is in uppercase, but all other letters must be lowercase. It's not enough to only make the first letter uppercase. You must also force the remaining letters to be lowercase. Converting the string `'mcCloud'` to title case doesn't result in `'McCloud'` but rather `'Mccloud'`.

There is also a boundary condition you should be aware of when looking at the "previous index" in the `for` loop. You can easily calculate the previous index from the index `i` with the expression `i - 1`, but there's a catch: when `i` is `0`, this results in `-1` which refers to the last index of the string. Your code must explicitly make sure you aren't checking the previous index for the first index of the string, because there is no previous index in that case.

Now try to write a solution based on the information in the previous sections. If you still have trouble solving this exercise, read the **Solution Template** section for additional hints.

Solution Template

Try to first write a solution from scratch. But if you have difficulty, you can use the following partial program as a starting place. Copy the following code from *https://inventwithpython.com/titlecase-template.py* and paste it into your code editor. Replace the underscores with code to make a working program:

```
def getTitleCase(text):
    # Create a titledText variable to store the titlecase text:
    titledText = ____
    # Loop over every index in text:
    for i in range(len(____)):
        # The character at the start of text should be uppercase:
        if i == ____:
            titledText += text[i].____()
        # If the character is a letter and the previous character is
        # not a letter, make it uppercase:
        elif text[____].isalpha() and not text[i - ____].isalpha():
            titledText += text[____].upper()
        # Otherwise, make it lowercase:
        else:
            titledText += text[i].____()
    # Return the titled cased string:
    return titledText
```

The complete solution for this exercise is given in Appendix A and *https://inventwithpython.com/titlecase.py*. You can view each step of this program as it runs under a debugger at *https://inventwithpython.com/titlecase-debug/*.

EXERCISE #36: REVERSE STRING

```
reverseString('Hello')  →  'olleH'
```

Strings are immutable in the Python language, meaning you can't modify their characters the way you can modify the items in a list. For example, if you tried to change `'Rat'` to `'Ram'` with the assignment statement `'Rat'[2] = 'm'`, you would receive a `TypeError: 'str' object does not support item assignment` error message. On the other hand, if you store a string 'Rat' in a variable named animal, the assignment statement animal = 'Ram' isn't modifying the 'Rat' string but rather making animal refer to an entirely new string, 'Ram'.

We can modify an existing string is to create a list of single-character strings, modify the list, and then create a new string from the list. Enter the following into the interactive shell:

```
>>> animal = 'Rat'
>>> animal = list(animal)
>>> animal
['R', 'a', 't']
>>> animal[2] = 'm'
>>> animal
['R', 'a', 'm']
>>> animal = ''.join(animal)
>>> animal
'Ram'
```

We'll use this technique to reverse the characters in a string.

Exercise Description

Write a `reverseString()` function with a `text` parameter. The function should return a string with all of `text`'s characters in reverse order. For example, `reverseString('Hello')` returns `'olleH'`. The function should not alter the casing of any letters. And, if `text` is a blank string, the function returns a blank string.

These Python `assert` statements stop the program if their condition is `False`. Copy them to the bottom of your solution program. Your solution is correct if the following `assert` statements' conditions are all `True`:

```
assert reverseString('Hello') == 'olleH'
assert reverseString('') == ''
```

```
assert reverseString('aaazzz') == 'zzzaaa'
assert reverseString('xxxx') == 'xxxx'
```

Try to write a solution based on the information in this description. If you still have trouble solving this exercise, read the **Solution Design and Special Cases and Gotchas** sections for additional hints.

Prerequisite concepts: lists, `list()`, `for` loops, `range()`, `len()`, integer division, indexes, swapping values, `join()`

Solution Design

Instead of building up a string from the characters in `text` in reverse order, let's make our function first convert the string in `text` into a list of single-character strings. Python's `list()` function does when we pass it a string. For example, `list('Hello')` returns the list value `['H', 'e', 'l', 'l', 'o']`. We can assign this list as the new value of `text`.

Once we have the characters of `text` in a list, create a `for` loop that loops over the first half of the list's indexes. This can be calculated from the expression `len(text) // 2`. We want to replace the character at each index with the character at the "mirror" index in the second half of the list. The mirror of the first index is the last index, the mirror of the second index is the second to last index, the mirror of the third index is the third to last index, and so on.

To calculate the mirror of an index `i` in `text`, you would want `len(text) - 1 - i`. For example, the mirror of index `0` is `len(text) - 1 - 0`, the mirror of index `1` is `len(text) - 1 - 1`, the mirror of index `2` is `len(text) - 1 - 2`, and so on.

Python's assignment statement allows you to swap two values simultaneously. For example, the assignment statement `myList[0], myList[5] = myList[5], myList[0]` swaps the values at indexes `0` and `5` in a hypothetical `myList` variable.

Figure 36-1 shows the characters of a hypothetical 12-item list made from `'Hello, world'` being swapped until the string is reversed.

Figure 36-1: The process of reversing a list of single-character strings by swapping their mirror indexes.

Finally, the `join()` string method creates a string from the text list with the instruction `''.join(text)`. This is the string the `reverseString()` function should return.

Special Cases and Gotchas

You may think that you need two different algorithms depending on if the string to reverse has an odd or even number of characters. However, it turns out this doesn't matter. If the `text` string has an odd number of characters, like the 5 characters in the list `['H', 'e', 'l', 'l', 'o']`, then swapping the middle character at index `2` with itself doesn't change the list.

Now try to write a solution based on the information in the previous sections. If you still have trouble solving this exercise, read the **Solution Template** section for additional hints.

Solution Template

Try to first write a solution from scratch. But if you have difficulty, you can use the following partial program as a starting place. Copy the following code from *https://invpy.com/reversestring-template.py* and paste it into your code editor. Replace the underscores with code to make a working program:

```
def reverseString(text):
    # Convert the text string into a list of character strings:
    text = ____(text)
    # Loop over the first half of indexes in the list:
    for i in range(len(____) // ____):
        # Swap the values of i and it's mirror index in the second
        # half of the list:
```

```
        mirrorIndex = len(text) - ____ - ____
        text[i], text[mirrorIndex] = text[____], text[____]
    # Join the list of strings into a single string and return it:
    return ''.join(text)
```

The complete solution for this exercise is given in Appendix A and *https://invpy.com/reversestring.py*. You can view each step of this program as it runs under a debugger at *https://invpy.com/reversestring-debug/*.

EXERCISE #37: CHANGE MAKER

```
makeChange(30)  →  {'quarters': 1, 'nickels': 1}
```

American currency has coins in the denominations of 1 (pennies), 5 (nickels), 10 (dimes), and 25 cents (quarters). Imagine that we were programming a cash register to dispense correct change. In this exercise, we would need to calculate the number of each coin for a given amount of change.

Exercise Description

Write a `makeChange()` function with an `amount` parameter. The `amount` parameter contains an integer of the number of cents to make change for. For example, `30` would represent 30 cents and `125` would represent $1.25. This function should return a dictionary with keys `'quarters'`, `'dimes'`, `'nickels'`, and `'pennies'`, where the value for a key is an integer of the number of this type of coin.

The value for a coin's key should never be `0`. Instead, the key should not be present in the dictionary. For example, `makeChange(5)` should return `{'nickels': 1}` and not `{'quarters': 0, 'dimes': 0, 'nickels': 1, 'pennies': 0}`.

For example, `makeChange(30)` would returns the dictionary `{'quarters': 1, 'nickels': 5}` to represent the coins used for 30 cents change. The function should use the minimal number of coins. For example, `makeChange(10)` should return `{'dimes': 1}` and not `{'nickels': 2}`, even though they both add up to 10 cents.

These Python `assert` statements stop the program if their condition is `False`. Copy them to the bottom of your solution program. Your solution is correct if the following `assert` statements' conditions are all `True`:

```
assert makeChange(30) == {'quarters': 1, 'nickels': 1}
assert makeChange(10) == {'dimes': 1}
assert makeChange(57) == {'quarters': 2, 'nickels': 1, 'pennies': 2}
assert makeChange(100) == {'quarters': 4}
assert makeChange(125) == {'quarters': 5}
```

Try to write a solution based on the information in this description. If you still have trouble solving this exercise, read the **Solution Design and Special Cases and Gotchas** sections for additional hints.

Prerequisite concepts: modulo operator, integer division

Solution Design

First, our `makeChange()` function should create an empty dictionary in a variable named `change` to store the results to return. Next, we need to determine the amount of each type of coin, starting with the largest denominations (25-cent quarters) to the smallest (1-cent pennies). This way, we don't accidentally use more than the minimum amount of coins by determining we need, for example, two 5-cent nickels instead of one 10 cent dime.

Let's start with quarters. Before doing any calculation, if the amount of change to make is less than 25, then we can skip this calculation entirely since there are zero quarters. Otherwise, if the amount of change to make is divisible by 25, say the `amount` parameter is `125`, then we can determine the number of quarters by dividing `amount` by `25`: `125 / 25` evaluates to `5.0`.

However, if it isn't divisible by 25, our result will have a fractional part: `135 / 25` evaluates to `5.4`. We can only add whole numbers of quarters to our change, not 0.4 quarters. Using the `//` integer division operator, we can ensure that we only put whole numbers of coins into our change dictionary: both `125 // 25` and `135 // 25` evaluate to `5`.

To deduct the amount of change held by the quarters, we can set `change` to the amount remaining after removing 25 cent increments. The word "remaining" hints that we should use the `%` modulo operator. For example, if we need to make 135 cents of change and use 5 quarters for 125 of those cents, we would need to use other coins for the `135 % 25` or `10` remaining cents.

This handles calculating the number of quarters used to make change. We would then copy-paste this code and make modifications for dimes, nickels, and pennies (in that order). When we finish processing the number of pennies, the `amount` parameter will be `0` and the `change` dictionary will contain the correct amounts of each coin.

Special Cases and Gotcha

Because this exercise specifies that the change should be made from the minimal number of coins, there is only one correct answer for any given amount of change. The most important thing to get right in this algorithm is to calculate the coins in order from largest to smallest, otherwise you'll end up in a situation where, say, you use two nickels instead of one dime.

Also, be sure to use the `//` integer division operator instead of the `/` division operator to calculate the number of coins. Python's regular division operator evaluates to floating-point numbers, which may contain a fractional part. (Even if it doesn't, Python still evaluates the result to a float: `125 / 25` evaluates to the float `5.0` and not the integer `5`.) Integer division always evaluates to integers and doesn't have this potential problem.

Because pennies represent 1 cent, you can set the number of pennies in change to whatever the `amount` parameter is at the end of the function as long as it is greater than zero.

Now try to write a solution based on the information in the previous sections. If you still have trouble solving this exercise, read the **Solution Template** section for additional hints.

Solution Template

Try to first write a solution from scratch. But if you have difficulty, you can use the following partial program as a starting place. Copy the following code from *https://invpy.com/makechange-template.py* and paste it into your code editor. Replace the underscores with code to make a working program:

```
def makeChange(amount):
    # Create a dictionary to keep track of how many of each coin:
    change = ____

    # If the amount is enough to add quarters, add them:
    if amount >= ____:
        change['quarters'] = amount // ____
        # Reduce the amount by the value of the quarters added:
        amount = amount % ____
    # If the amount is enough to add dimes, add them:
    if amount >= ____:
        change['dimes'] = ____ // ____
        # Reduce the amount by the value of the dimes added:
        amount = ____ % ____
    # If the amount is enough to add nickels, add them:
    if amount >= ____:
        change['nickels'] = ____ // ____
        # Reduce the amount by the value of the nickels added:
        amount = ____ % ____
    # If the amount is enough to add pennies, add them:
    if amount >= 1:
        change[____] = amount

    return change
```

The complete solution for this exercise is given in Appendix A and *https://invpy.com/makechange.py*. You can view each step of this program as it runs under a debugger at *https://invpy.com/makechange-debug/*.

EXERCISE #38: RANDOM SHUFFLE

```
shuffle([1, 2, 3, 4, 5])  →  [3, 1, 4, 5, 2]
```

A random shuffle algorithm puts the values in a list into a random order, like shuffling a deck of cards. This algorithm produces a new *permutation*, or ordering, of the values in the list. The algorithm works by looping over each value in the list and randomly determining a new index with which to swap it. As a result, the values in the list are in random order.

For a list of n values, there are n! ("n factorial") possible permutations. For example, a 10-value list has 10! or 10 × 9 × 8 × 7 × 6 × 5 × 4 × 3 × 2 × 1 or 3,628,800 possible ways to order them.

This exercise modifies the list passed to it *in-place*, rather than creating a new list and returning it. Because lists are mutable objects in Python, modifications made to a parameter are actually modifying the original object passed to the function call for that parameter. For example, enter the following into the interactive shell:

```
>>> someList = [1, 2, 3]  # Let's create a list object.
>>> def someFunc(someParam):
...     someParam[0] = 'dog'  # This is changing the list in-place.
...     someParam.append('xyz')  # This is changing the list in-place.
...
>>> someList
[1, 2, 3]
>>> someFunc(someList)  # Pass the list as the argument.
>>> someList  # Note that the list object has been modified by the function.
['dog', 2, 3, 'xyz']
```

Notice that the `someList` list is passed as the argument for the `someParam` parameter of the `someFunc()` function. This function modifies `someParam` (which refers to the same list object that the `someList` variable refers to), so these modifications are still there after the function returns. The `someFunc()` function isn't returning a new list to replace `someList`; it's modifying `someList` in-place.

In Python, only mutable objects (such as lists, dictionaries, and sets) can be modified in-place. Immutable objects (such a strings, integers, tuples, and frozen sets) can't be modified in-place.

Exercise Description

Write a `shuffle()` function with a `values` parameter set to a list of values. The function doesn't return anything, but rather it sets each value in the list to a random index. The resulting

shuffled list must contain the same values as before but in random order.

This exercise asks you to implement a function identical to Python's `random.shuffle()` function. As such, avoid using this function in your solution as it'd defeat the purpose of the exercise.

These Python `assert` statements stop the program if their condition is `False`. Copy them to the bottom of your solution program. Your solution is correct if the following `assert` statements' conditions are all `True`:

```
random.seed(42)
# Perform this test ten times:
for i in range(10):
    testData1 = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
    shuffle(testData1)
    # Make sure the number of values hasn't changed:
    assert len(testData1) == 10
    # Make sure the order has changed:
    assert testData1 != [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
    # Make sure that when re-sorted, all the original values are there:
    assert sorted(testData1) == [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# Make sure an empty list shuffled remains empty:
testData2 = []
shuffle(testData2)
assert testData2 == []
```

Try to write a solution based on the information in this description. If you still have trouble solving this exercise, read the **Solution Design and Special Cases and Gotchas** sections for additional hints.

Prerequisite concepts: `import` statements, `random` module, `randint()`, `for` loops, `range()`, `len()`, swapping values

Solution Design

The solution is surprisingly straightforward. A `for` loop can loop over every index in the list. On each iteration, the code in the loop selects a random index. Then it swaps the values at the current iteration's index and the random index.

If the random index is the same as the current iteration's index, this is fine: a random shuffling can include values at their original location. This isn't somehow "less random" than any other permutation. If the random index is a repeat of an index that has previously been swapped, this is fine as well. Shuffling a value to a random location twice isn't any more or less shuffled than moving a value to a random location once.

Special Cases and Gotchas

Take care that your `shuffle()` function doesn't add or remove any values to the list; it should only rearrange the values already in the list. Because the `shuffle()` function modifies the values list argument in-place, it doesn't need to return anything. There shouldn't be a `return` statement anywhere in the function. In Python, all functions technically return a value; it's just that functions with no `return` statement return the value `None`.

When selecting a random index, select only an index within the range of the list's indexes. This means you should select an index from `0` up to, but not including, the length of the list. When calling

`random.randint()` to generate this random index, you'll want to use `0` and `len(values) - 1` to represent this range, and not `0` and `len(values)`.

Now try to write a solution based on the information in the previous sections. If you still have trouble solving this exercise, read the **Solution Template** section for additional hints.

Solution Template

Try to first write a solution from scratch. But if you have difficulty, you can use the following partial program as a starting place. Copy the following code from *https://invpy.com/randomshuffle-template.py* and paste it into your code editor. Replace the underscores with code to make a working program:

```
# Import the random module for its randint() function.
import random

def shuffle(values):
    # Loop over the range of indexes from 0 up to the length of the list:
    for i in range(____(values)):
        # Randomly pick an index to swap with:
        swapIndex = random.randint(0, len(____) - ____)
        # Swap the values between the two indexes:
        values[i], values[swapIndex] = values[____], values[____]
```

The complete solution for this exercise is given in Appendix A and *https://invpy.com/randomshuffle.py*. You can view each step of this program as it runs under a debugger at *https://invpy.com/randomshuffle-debug/*.

EXERCISE #39: COLLATZ SEQUENCE

```
collatz(10)  →  [10, 5, 16, 8, 4, 2, 1]
```

The Collatz Sequence also called the $3n + 1$ problem, is a simple but mysterious numeric sequence that has remained unsolved by mathematicians. It has four rules:

- Begin with a positive, nonzero integer called *n*.
- If *n* is 1, the sequence terminates.
- If *n* is even, the next value of *n* is $n / 2$.
- If *n* is odd, the next value of *n* is $3n + 1$.

For example, if the starting integer is 10, that number is even so the next number is 10 / 2, or 5. 5 is odd, so the next number is $3 \times 5 + 1$, or 16. 16 is even, so the next number is 8, which is even so the next number is 4, then 2, then 1. At 1, the sequence stops. The entire Collatz Sequence starting at 10 is: 10, 5, 16, 8, 4, 2, 1

Mathematicians have been unable to prove if every starting integer eventually terminates. This gives the Collatz Sequence the description of "the simplest impossible math problem." However, in this exercise, all you need to do is calculate the sequence of numbers for a given starting integer.

Exercise Description

Write a function named `collatz()` with a `startingNumber` parameter. The function returns a list of integers of the Collatz sequence that `startingNumber` produces. The first integer in this list must be `startingNumber` and the last integer must be `1`.

Your function should check if `startingNumber` is an integer less than 1, and in that case, return an empty list.

These Python `assert` statements stop the program if their condition is `False`. Copy them to the bottom of your solution program. Your solution is correct if the following `assert` statements' conditions are all `True`:

```
assert collatz(0) == []
assert collatz(10) == [10, 5, 16, 8, 4, 2, 1]
assert collatz(11) == [11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1]
assert collatz(12) == [12, 6, 3, 10, 5, 16, 8, 4, 2, 1]
```

```
assert len(collatz(256)) == 9
assert len(collatz(257)) == 123
import random
random.seed(42)
for i in range(1000):
    startingNum = random.randint(1, 10000)
    seq = collatz(startingNum)
    assert seq[0] == startingNum # Make sure it includes the starting number.
    assert seq[-1] == 1  # Make sure the last integer is 1.
```

Try to write a solution based on the information in this description. If you still have trouble solving this exercise, read the **Solution Design and Special Cases and Gotchas** sections for additional hints.

Prerequisite concepts: lists, `while` loops, modulo operator, integer division, `append()`

Solution Design

The function only needs a variable to keep track of the current number, which we can call `num`, and a variable to hold the sequence of values, which we can call `sequence`. At the start of the function, set `num` to the integer in `startingNumber` parameter and `sequence` to `[num]`. We can use a `while` loop that continues to loop as long as the `num` is not `1`. On each iteration of the loop, the next value for `num` is calculated based on whether `num` is currently odd or even. You can use the modulo 2 technique from Exercise #3, "Odd & Even" to determine this: if `num % 2` evaluates to `0` then `num` is even and if it evaluates to `1` then `num` is odd. After this, append `num` to the end of the `sequence` list.

If `num` is exactly `1`, then the `while` loop stops looping and the function can return `sequence`.

Special Cases and Gotchas

The only special case is if the `startingNumber` parameter is less than 1, in which case there is no sequence and the function should return an empty list `[]`.

Now try to write a solution based on the information in the previous sections. If you still have trouble solving this exercise, read the **Solution Template** section for additional hints.

Solution Template

Try to first write a solution from scratch. But if you have difficulty, you can use the following partial program as a starting place. Copy the following code from *https://invpy.com/collatzsequence-template.py* and paste it into your code editor. Replace the underscores with code to make a working program:

```
def collatz(startingNumber):
    # If the starting number is 0 or negative, return an empty list:
    if ____ < 1:
        return ____

    # Create a list to hold the sequence, beginning with the starting number:
    sequence = [____]
    num = ____
    # Keep looping until the current number is 1:
```

```
    while num ____ 1:
        # If odd, the next number is 3 times the current number plus 1:
        if num % 2 == ____:
            num = 3 * num + 1
        # If even, the next number is half the current number:
        else:
            num = num // ____
        # Record the number in the sequence list:
        sequence.append(____)

    # Return the sequence of numbers:
    return ____
```

The complete solution for this exercise is given in Appendix A and *https://invpy.com/collatzsequence.py*. You can view each step of this program as it runs under a debugger at *https://invpy.com/collatzsequence-debug/*.

Further Reading

You can find out more about the Collatz sequence on Wikipedia at *https://en.wikipedia.org/wiki/Collatz_conjecture*. There are videos on YouTube about the sequence on the Veritasium channel titled "The Simplest Math Problem No One Can Solve - Collatz Conjecture" at *https://youtu.be/094y1Z2wpJg* and and the Numberphile channel titled "UNCRACKABLE? The Collatz Conjecture" at *https://youtu.be/5mFpVDpKX70*.

EXERCISE #40: MERGING TWO SORTED LISTS

```
mergeTwoLists([1, 3, 6], [5, 7, 8, 9])  →  [1, 3, 5, 6, 7, 8, 9]
```

One of the most efficient sorting algorithms is the merge sort algorithm. Merge sort has two phases: the dividing phase and the merge phase. We won't dive into this advanced algorithm in this book. However, we can write code for the second half: merging two pre-sorted lists of integers into a single sorted list.

Exercise Description

Write a `mergeTwoLists()` function with two parameters `list1` and `list2`. The lists of numbers passed for these parameters are already in sorted order from smallest to largest number. The function returns a single sorted list of all numbers from these two lists.

You could write this function in one line of code by using Python's `sorted()` function:

```
return sorted(list1 + list2)
```

But this would defeat the purpose of the exercise, so don't use the `sorted()` function or `sort()` method as part of your solution.

These Python `assert` statements stop the program if their condition is `False`. Copy them to the bottom of your solution program. Your solution is correct if the following `assert` statements' conditions are all `True`:

```
assert mergeTwoLists([1, 3, 6], [5, 7, 8, 9]) == [1, 3, 5, 6, 7, 8, 9]
assert mergeTwoLists([1, 2, 3], [4, 5]) == [1, 2, 3, 4, 5]
assert mergeTwoLists([4, 5], [1, 2, 3]) == [1, 2, 3, 4, 5]
assert mergeTwoLists([2, 2, 2], [2, 2, 2]) == [2, 2, 2, 2, 2, 2]
assert mergeTwoLists([1, 2, 3], []) == [1, 2, 3]
assert mergeTwoLists([], [1, 2, 3]) == [1, 2, 3]
```

Try to write a solution based on the information in this description. If you still have trouble solving this exercise, read the **Solution Design and Special Cases and Gotchas** sections for additional hints.

Prerequisite concepts: lists, `while` loops, Boolean operators, `append()`, `for` loops, `range()` with two arguments, `len()`

Solution Design

The lists of integers, already in sorted order, are passed to the function as parameters `list1` and `list2`. The algorithm begins with two variables, `i1` and `i2`, which both begin at index `0` of their respective lists. We also create a blank list in a variable named `result` which stores the merged results of the two lists.

Inside of a `while` loop, the code does the following:

- Look at the numbers that `i1` and `i2` point to.
- Append the smaller of the two numbers to `result`.
- If `i1`'s number was appended, increment `i1` to point to the next number in `list1`. Otherwise, increment `i2` to point to the next number in `list2`.
- Repeat until either `i1` or `i2` has gone past the end of their list.

For example, Figure 40-1 shows the first three iterations of the loop when merging lists `[1, 3, 6]` and `[5, 7, 8, 9]`.

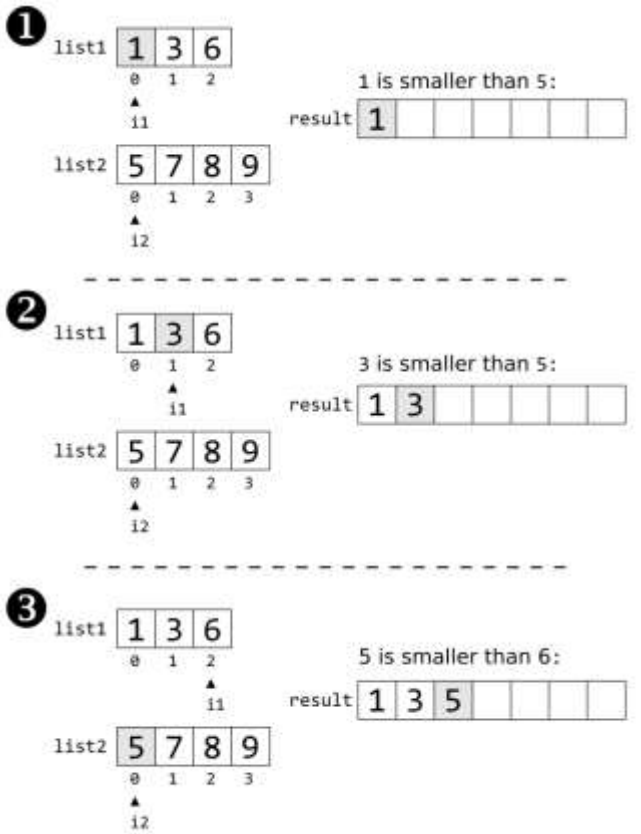


Figure 40-1: The first three iterations of the loop that merges two lists.

Think of this code as like an asymmetrical zipper: the `i1` and `i2` variables keep moving right along their respective lists, appending their values to result. When either `i1` or `i2` reaches the end of their list, the rest of the other list is appended to `result`. This `result` list contains all of the numbers in `list1` and `list2` in sorted order, so the function returns it.

Special Cases and Gotchas

Keep in mind that while the `list1` and `list2` parameters must be sorted lists, they aren't required to be the same length.

If either of these list arguments to `mergeTwoLists()` isn't sorted, the function will return a merged list that is also not in sorted order. For this exercise, however, we'll assume that `mergeTwoLists()` is always called with valid arguments.

Now try to write a solution based on the information in the previous sections. If you still have trouble solving this exercise, read the **Solution Template** section for additional hints.

Solution Template

Try to first write a solution from scratch. But if you have difficulty, you can use the following partial program as a starting place. Copy the following code from *https://invpy.com/mergetwolists-template.py* and paste it into your code editor. Replace the underscores with code to make a working program:

```
def mergeTwoLists(list1, list2):
    # Create an empty list to hold the final sorted results:
    result = ____

    # Start i1 and i2 at index 0, the start of list1 and list2:
    i1 = ____
    i2 = ____

    # Keeping moving up i1 and i2 until one reaches the end of its list:
    while i1 < len(____) and ____ < len(list2):
        # Add the smaller of the two current items to the result:
        if list1[____] < list2[____]:
            # Add list1's current item to the result:
            result.append(____[i1])
            # Increment i1:
            i1 += ____
        else:
            # Add list2's current item to the result:
            result.append(____[i2])
            # Increment i2:
            i2 += ____

    # If i1 is not at the end of list1, add the remaining items from list1:
    if i1 < len(____):
        for j in range(i1, len(list1)):
            result.append(____[j])
    # If i2 is not at the end of list2, add the remaining items from list2:
    if i2 < len(____):
        for j in range(i2, len(list2)):
            result.append(____[j])

    # Return the merged, sorted list:
    return result
```

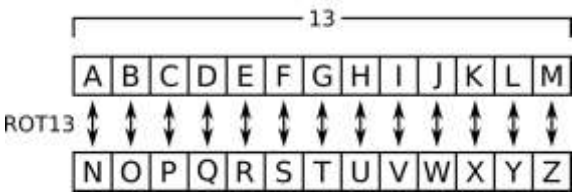
The complete solution for this exercise is given in Appendix A and *https://invpy.com/mergetwolists.py*. You can view each step of this program as it runs under a debugger at *https://invpy.com/mergetwolists-debug/*.

Further Reading

Merge sort uses this "merge two sorted lists into a single sorted list" behavior as a step in its algorithm. You can learn more about merge sort and other recursive algorithms from my book, "The Recursive Book of Recursion." The full book is freely available under a Creative Commons license at *https://inventwithpython.com/recursion/*.

EXERCISE #41: ROT 13 ENCRYPTION

```
rot13('Hello, world!')  →  'Uryyb, jbeyq!'
rot13('Uryyb, jbeyq!')  →  'Hello, world!'
```



ROT 13 is a simple encryption *cipher*. The name "ROT 13" is short for "rotate 13." It encrypts by replacing letters with letters that appear 13 characters down the alphabet: A is replaced with N, B is replaced with O, C is replaced with P, and so on. If this rotation of 13 letters goes passed the end of the alphabet, it "wraps around" the Z and continues from the start of the alphabet. Thus, X is replaced with K, Y is replaced with L, Z is replaced with M, and so on. Non-letter characters are left unencrypted.

The benefit of ROT 13 is that you can decrypt the encrypted text by running it through ROT 13 encryption again. This rotates the letter 26 times, returning us to the original letter. So "Hello, world!" encrypts to "Uryyb, jbeyq!" which in turn encrypts to "Hello, world!" There is no decryption algorithm; you decrypt encrypted text by encrypting it again. The ROT 13 algorithm isn't secure for real-world cryptography. But it can be used to obfuscate text to prevent spoiling joke punch lines or puzzle solutions.

The following shows what each of the 26 letters encrypts to with ROT 13 once (from the top row to the middle row) and twice (from the middle row to the bottom row.)

```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
▼ ▼ ▼ ▼ ▼ ▼ ▼ ▼ ▼ ▼ ▼ ▼ ▼ ▼ ▼ ▼ ▼ ▼ ▼ ▼ ▼ ▼ ▼ ▼ ▼ ▼
N O P Q R S T U V W X Y Z A B C D E F G H I J K L M
▼ ▼ ▼ ▼ ▼ ▼ ▼ ▼ ▼ ▼ ▼ ▼ ▼ ▼ ▼ ▼ ▼ ▼ ▼ ▼ ▼ ▼ ▼ ▼ ▼ ▼
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
```

Exercise Description

Write a `rot13()` function with a `text` parameter that returns the ROT 13 encrypted version of `text`. Uppercase letters encrypt to uppercase letters and lowercase letters encrypt to lowercase letters. For example, `'HELLO, world!'` encrypts to `'URYYB, jbeyq!'` and `'hello, WORLD!'` encrypts to `'uryyb, JBEYQ!'`.

You may use the following Python functions and string methods as part of your solution: `ord()`, `chr()`, `isalpha()`, `islower()`, and `isupper()`.

These Python `assert` statements stop the program if their condition is `False`. Copy them to

the bottom of your solution program. Your solution is correct if the following `assert` statements' conditions are all `True`:

```
assert rot13('Hello, world!') == 'Uryyb, jbeyq!'
assert rot13('Uryyb, jbeyq!') == 'Hello, world!'
assert rot13(rot13('Hello, world!')) == 'Hello, world!'
assert rot13('abcdefghijklmnopqrstuvwxyz') == 'nopqrstuvwxyzabcdefghijklm'
assert rot13('ABCDEFGHIJKLMNOPQRSTUVWXYZ') == 'NOPQRSTUVWXYZABCDEFGHIJKLM'
```

Try to write a solution based on the information in this description. If you still have trouble solving this exercise, read the **Solution Design and Special Cases and Gotchas** sections for additional hints.

Prerequisite concepts: strings, `ord()`, `chr()`, `for` loops, Boolean operators, `islower()`, `isupper()`, augmented assignment operators

Solution Design

Instead of hard-coding every letter and its encrypted form, we can rely on each letter's Unicode code point integer. Code points were discussed in Exercise #7, "ASCII Table." The `ord()` and `chr()` functions discussed in Exercise #7, "ASCII Table" can translate from a letter string to integer and integer to letter string, respectively.

The function starts with an `encryptedText` variable set to an empty string that will store the encrypted result as we encrypt each character. A `for` loop can loop over the `text` parameter to encrypt each character. If this character isn't a letter, it's added to the end of `encryptedText` as-is without encryption.

Otherwise, we can pass the letter to `ord()` to obtain its Unicode code point as an integer. Uppercase letters A to Z have integers ranging from 65 up to and including 90. Lowercase letters a to z have integers ranging from 97 up to and including 122. We need to reduce this by 26 to "wrap around" to the start of the alphabet.

For example, the letter `'S'` has an integer `83` (because `ord('S')` returns `83`) but adding `83 + 13` gives us `96`, which is greater than the integer for Z (`ord('Z')` returns `90`). In this case, we must subtract `26`: `96 - 26` gives us the encrypted integer `70`, and `chr(70)` returns `'F'`. This is how we can determine that `'S'` encrypts to `'F'` in the ROT 13 cipher.

Note that while an uppercase `'Z'` has the Unicode code point `90`, the lowercase `'z'` has the Unicode code point `122`.

Special Cases and Gotchas

While you want to add `13` to the Unicode code point integers of both uppercase and lowercase letters, when you check if this addition results in a number larger than Z's Unicode code point, you must use the correct case of Z. Otherwise, your `rot13()` function may determine that the lowercase `'a'` (with integer `97`) is past uppercase `'Z'` (with integer `90`) because `97` is greater than `90`. You must compare lowercase rotated letters with `122` (the integer of lowercase `'z'`) and uppercase rotated letters with `90` (the integer of uppercase `'Z'`).

All non-letter characters such as numbers, spaces, and punctuation marks are added to the encrypted text unmodified. Be sure that your `rot13()` function doesn't accidentally drop them from the returned string.

Now try to write a solution based on the information in the previous sections. If you still have trouble solving this exercise, read the **Solution Template** section for additional hints.

Solution Template

Try to first write a solution from scratch. But if you have difficulty, you can use the following partial program as a starting place. Copy the following code from *https://invpy.com/rot13-template.py* and paste it into your code editor. Replace the underscores with code to make a working program:

```
def rot13(text):
    # Create an encryptedText variable to store the encrypted string:
    encryptedText = ____
    # Loop over each character in the text:
    for character in text:
        # If the character is not a letter, add it as-is to encryptedText:
        if not character.____():
            encryptedText += ____
        # Otherwise calculate the letter's "rotated 13" letter:
        else:
            rotatedLetterOrdinal = ____(character) + 13
            # If adding 13 pushes the letter past Z, subtract 26:
            if ____.islower() and rotatedLetterOrdinal > ____:
                rotatedLetterOrdinal -= ____
            if ____.isupper() and rotatedLetterOrdinal > ____:
                rotatedLetterOrdinal -= ____

            # Add the encrypted letter to encryptedText:
            encryptedText += ____(rotatedLetterOrdinal)

    # Return the encrypted text:
    return encryptedText
```

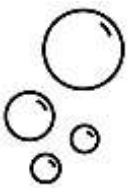
The complete solution for this exercise is given in Appendix A and *https://invpy.com/rot13.py*. You can view each step of this program as it runs under a debugger at *https://invpy.com/rot13-debug/*.

Further Reading

If you are interested in writing Python programs for encryption algorithms and code breaking, my book "Cracking Codes with Python" is freely available under a Creative Commons license at *https://inventwithpython.com/cracking/*.

EXERCISE #42: BUBBLE SORT

```
bubbleSort([2, 0, 4, 1, 3])  →  [0, 1, 2, 3, 4]
```



Bubble sort is often the first sorting algorithm taught to computer science students. While it is too inefficient for use in real-world software, the algorithm is easy to understand. In this last exercise of the book, we'll implement this basic sorting algorithm.

Exercise Description

Write a `bubbleSort()` function with a list parameter named `numbers`. The function rearranges the values in this list in-place. The function also returns the now-sorted list. There are many sorting algorithms, but this exercise asks you to implement the bubble sort algorithm.

The objective of this exercise is to write a sorting algorithm, so avoid using Python's `sort()` method or `sorted()` function as that would defeat the purpose of the exercise.

These Python `assert` statements stop the program if their condition is `False`. Copy them to the bottom of your solution program. Your solution is correct if the following `assert` statements' conditions are all `True`:

```
assert bubbleSort([2, 0, 4, 1, 3]) == [0, 1, 2, 3, 4]
assert bubbleSort([2, 2, 2, 2]) == [2, 2, 2, 2]
```

Try to write a solution based on the information in this description. If you still have trouble solving this exercise, read the **Solution Design and Special Cases and Gotchas** sections for additional hints.

Prerequisite concepts: lists, `for` loops, `range()` with two arguments, nested loops, swapping values

Solution Design

The bubble sort algorithm compares every pair of indexes and swaps their values so that the larger value comes later in the list. As the algorithm runs, the larger numbers "bubble up" towards the end, hence the algorithm's name. We'll use variables named `i` and `j` to track the two indexes whose values should be compared with each other. The pattern behind the movements of `i` and `j` are easier to see when visually laid out, as in Figure 42-1 which uses a 5-item `numbers` list as an example:

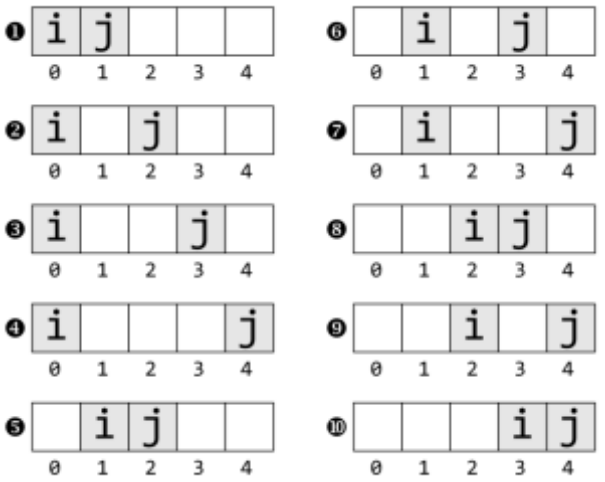


Figure 42-1: The pattern of i and j's movement: j starts after i and moves to the right, and when it reaches the end, i moves right once and j starts after i again.

Notice the similarity between the movement of `i` and `j` to the nested `for` loops in Project #26 "Handshakes." As the algorithm runs, `j` starts after `i` and moves to the right, and when it reaches the end, `i` moves right once and `j` starts after `i` again.

If you look at the overall range of `i` and `j`, you'll see that `i` starts at index `0` and ends at the second to last index. Meanwhile, `j` starts at the index after `i` and ends at the last index. This means our nested `for` loops over the `numbers` list parameter would look like this:

```
for i in range(len(numbers) - 1):
    for j in range(i, len(numbers)):
```

Inside the inner loop, the numbers at indexes `i` and `j` are compared, and if the number at index `i` is larger than the number at index `j`, they are swapped. Figure 42-2 shows the state of a list `[8, 2, 9, 6, 3]` as the bubble sort algorithm swaps the two numbers after being compared at each step.

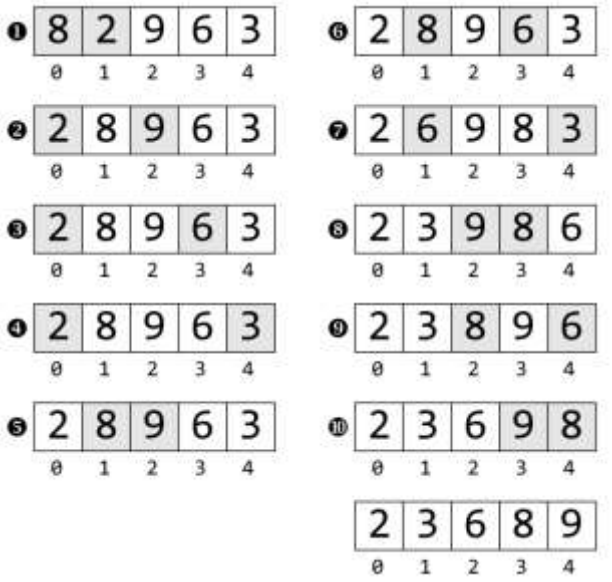


Figure 42-2: The steps of the bubble sort algorithm as it sorts [8, 2, 9, 6, 3].

At the end of these two nested `for` loops, the numbers in the list will have been swapped into sorted order.

Special Cases and Gotchas

Sorting algorithms are an excellent introduction to the computer science topic of *data structures and algorithms*. And bubble sort is a good introduction to sorting algorithms. But the chief weakness of bubble sort is that it's incredibly inefficient. While it can quickly sort lists of a few dozen or few hundred values, it becomes infeasible for sorting lists of thousands or millions of values. For this reason, real-world applications never use bubble sort.

Now try to write a solution based on the information in the previous sections. If you still have trouble solving this exercise, read the **Solution Template** section for additional hints.

Solution Template

Try to first write a solution from scratch. But if you have difficulty, you can use the following partial program as a starting place. Copy the following code from *https://invpy.com/bubblesort-template.py* and paste it into your code editor. Replace the underscores with code to make a working program:

```
def bubbleSort(numbers):
    # The outer loop loops i over all but the last number:
    for i in range(len(____) - ____):
        # The inner loop loops j starting at i to the last number:
        for j in range(____, len(____)):
            # If the number at i is greater than the number at j, swap them:
            if numbers[i] ____ numbers[j]:
                numbers[i], numbers[j] = numbers[____], numbers[____]
    # Return the now-sorted list:
    return numbers
```

The complete solution for this exercise is given in Appendix A and *https://invpy.com/bubblesort.py*. You can view each step of this program as it runs under a debugger at *https://invpy.com/bubblesort-debug/*.

Further Reading

If you want to see what a first-year computer science student would study in a data structures and algorithms course, Coursera has a free online course called "Algorithmic Toolbox" at *https://www.coursera.org/learn/algorithmic-toolbox*.

APPENDIX A: SOLUTIONS

As long as your programs pass all the `assert` statements or match the output given in the exercise, they don't have to be identical to my solutions. There's always several ways to write code. This appendix contains the complete solutions that were partially given in the **Solution Template** sections of each exercise.

I highly recommend attempting to solve these exercises on your own instead of immediately jumping to these solutions, even if you have to struggle with them for a while. More important than the knowledge of these solutions is the practice that attempting to solve them gives you. However, if you feel stuck and cannot move on, reading the solution program for an exercise can give you insights for how to solve the other exercises in this book.

Exercise #1: Hello, World!

```
# Print "Hello, world!" on the screen:
print('Hello, world!')
# Ask the user for their name:
print('What is your name?')
# Get the user's name from their keyboard input:
name = input()
# Greet the user by their name:
print('Hello, ' + name)
```

Exercise #2: Temperature Conversion

```
def convertToFahrenheit(degreesCelsius):
    # Calculate and return the degrees Fahrenheit:
    return degreesCelsius * (9 / 5) + 32


def convertToCelsius(degreesFahrenheit):
    # Calculate and return the degrees Celsius:
    return (degreesFahrenheit - 32) * (5 / 9)
```

Exercise #3: Odd & Even

```
def isOdd(number):
    # Return whether number mod 2 is 1:
    return number % 2 == 1

def isEven(number):
    # Return whether number mod 2 is 0:
    return number % 2 == 0
```

Exercise #4: Area & Volume

```
def area(length, width):
    # Return the product of the length and width:
    return length * width

def perimeter(length, width):
    # Return the sum of the length twice and the width twice:
    return length * 2 + width * 2

def volume(length, width, height):
    # Return the product of the length, width, and height:
    return length * width * height

def surfaceArea(length, width, height):
    # Return the sum of the area of each of the six sides:
    return ((length * width) + (length * height) + (width * height)) * 2
```

Exercise #5: Fizz Buzz

```
def fizzBuzz(upTo):
    # Loop from 1 up to (and including) the upTo parameter:
    for number in range(1, upTo + 1):
        # If the loop number is divisible by 3 and 5, print 'FizzBuzz':
        if number % 3 == 0 and number % 5 == 0:
            print('FizzBuzz', end=' ')
        # Otherwise the loop number is divisible by only 3, print 'Fizz':
        elif number % 3 == 0:
            print('Fizz', end=' ')
        # Otherwise the loop number is divisible by only 5, print 'Buzz':
        elif number % 5 == 0:
            print('Buzz', end=' ')
        # Otherwise, print the loop number:
        else:
            print(number, end=' ')
```

Exercise #6: Ordinal Suffix

```
def ordinalSuffix(number):
    numberStr = str(number)

    # 11, 12, and 13 have the suffix th:
    if numberStr[-2:] in ('11', '12', '13'):
        return numberStr + 'th'
    # Numbers that end with 1 have the suffix st:
    if numberStr[-1] == '1':
        return numberStr + 'st'
    # Numbers that end with 2 have the suffix nd:
    if numberStr[-1] == '2':
        return numberStr + 'nd'
    # Numbers that end with 3 have the suffix rd:
    if numberStr[-1] == '3':
        return numberStr + 'rd'
    # All other numbers end with th:
    return numberStr + 'th'
```

Alternate Solution:

```
def ordinalSuffix(number):
    # 11, 12, and 13 have the suffix th:
    if number % 100 in (11, 12, 13):
        return str(number) + 'th'
    # Numbers that end with 1 have the suffix st:
    if number % 10 == 1:
        return str(number) + 'st'
    # Numbers that end with 2 have the suffix nd:
    if number % 10 == 2:
        return str(number) + 'nd'
    # Numbers that end with 3 have the suffix rd:
    if number % 10 == 3:
        return str(number) + 'rd'
    # All other numbers end with th:
    return str(number) + 'th'
```

Exercise #7: ASCII Table

```
def printASCIITable():
    # Loop over integers 32 up to and including 126:
    for i in range(32, 127):
        # Print the integer and its ASCII text character:
        print(i, chr(i))
printASCIITable()
```

Exercise #8: Read Write File

```
def writeToFile(filename, text):
    # Open the file in write mode:
    with open(filename, 'w') as fileObj:
        # Write the text to the file:
        fileObj.write(text)

def appendToFile(filename, text):
    # Open the file in append mode:
    with open(filename, 'a') as fileObj:
        # Write the text to the end of the file:
        fileObj.write(text)

def readFromFile(filename):
    # Open the file in read mode:
    with open(filename) as fileObj:
        # Read all of the text in the file and return it as a string:
        return fileObj.read()
```

Exercise #9: Chess Square Color

```
def getChessSquareColor(column, row):
    # If the column and row is out of bounds, return a blank string:
    if column < 1 or column > 8 or row < 1 or row > 8:
        return ''
```

```
    # If the even/oddness of the column and row match, return 'white':
    if column % 2 == row % 2:
        return 'white'
    # If they don't match, then return 'black':
    else:
        return 'black'
```

Here is a second solution:

```
def getChessSquareColor(column, row):
    # If the column and row is out of bounds, return a blank string:
    if column < 1 or column > 8 or row < 1 or row > 8:
        return ''

    # If the even/oddness of the column and row match, return 'white':
    if not (column % 2 ^ row % 2):
        return 'white'
    # If they don't match, then return 'black':
    else:
        return 'black'
```

Exercise #10: Find and Replace

```
def findAndReplace(text, oldText, newText):
    replacedText = ''
    i = 0
    while i < len(text):
        # If index i in text is the start of the oldText pattern, add
        # the replacement text:
        if text[i:i + len(oldText)] == oldText:
            # Add the replacement text:
            replacedText += newText
            # Increment i by the length of oldText:
            i += len(oldText)
        # Otherwise, add the characters at text[i] and increment i by 1:
        else:
            replacedText += text[i]
            i += 1
    return replacedText
```

Exercise #11: Hours, Minutes, Seconds

```
def getHoursMinutesSeconds(totalSeconds):
    # If totalSeconds is 0, just return '0s':
    if totalSeconds == 0:
        return '0s'

    # Set hours to 0, then add an hour for every 3600 seconds removed from
    # totalSeconds until totalSeconds is less than 3600:
    hours = 0
    while totalSeconds >= 3600:
        hours += 1
        totalSeconds -= 3600

    # Set minutes to 0, then add a minute for every 60 seconds removed from
```

```
    # totalSeconds until totalSeconds is less than 60:
    minutes = 0
    while totalSeconds >= 60:
        minutes += 1
        totalSeconds -= 60

    # Set seconds to the remaining totalSeconds value:
    seconds = totalSeconds

    # Create an hms list that contains the string hour/minute/second amounts:
    hms = []
    # If there are one or more hours, add the amount with an 'h' suffix:
    if hours > 0:
        hms.append(str(hours) + 'h')
    # If there are one or more minutes, add the amount with an 'm' suffix:
    if minutes > 0:
        hms.append(str(minutes) + 'm')
    # If there are one or more seconds, add the amount with an 's' suffix:
    if seconds > 0:
        hms.append(str(seconds) + 's')

    # Join the hour/minute/second strings with a space in between them:
    return ' '.join(hms)
```

Exercise #12: Smallest & Biggest

```
def getSmallest(numbers):
    # If the numbers list is empty, return None:
    if len(numbers) == 0:
        return None

    # Create a variable that tracks the smallest value so far, and start
    # it off a the first value in the list:
    smallest = numbers[0]
    # Loop over each number in the numbers list:
    for number in numbers:
        # If the number is smaller than the current smallest value, make
        # it the new smallest value:
        if number < smallest:
            smallest = number
    # Return the smallest value found:
    return smallest
```

Exercise #13: Sum & Product

```
def calculateSum(numbers):
    # Start the sum result at 0:
    result = 0
    # Loop over all the numbers in the numbers parameter, and add them
    # to the running sum result:
    for number in numbers:
        result += number
    # Return the final sum result:
    return result
```

```
def calculateProduct(numbers):
    # Start the product result at 1:
    result = 1
    # Loop over all the numbers in the numbers parameter, and multiply
    # them by the running product result:
    for number in numbers:
        result *= number
    # Return the final product result:
    return result
```

Exercise #14: Average

```
def average(numbers):
    # Special case: If the numbers list is empty, return None:
    if len(numbers) == 0:
        return None

    # Start the total at 0:
    total = 0

    # Loop over each number in numbers:
    for number in numbers:
        # Add the number to the total:
        total += number

    # Get the average by dividing the total by how many numbers there are:
    return total / len(numbers)
```

Exercise #15: Median

```
def median(numbers):
    # Special case: If the numbers list is empty, return None:
    if len(numbers) == 0:
        return None

    # Sort the numbers list:
    numbers.sort()

    # Get the index of the middle number:
    middleIndex = len(numbers) // 2

    # If the numbers list has an even length, return the average of the
    # middle two numbers:
    if len(numbers) % 2 == 0:
        return (numbers[middleIndex] + numbers[middleIndex - 1]) / 2
    # If the numbers list has an odd length, return the middlemost number:
    else:
        return numbers[middleIndex]
```

Exercise #16: Mode

```
def mode(numbers):
    # Special case: If the numbers list is empty, return None:
    if len(numbers) == 0:
```

```
        return None

    # Dictionary with keys of numbers and values of how often they appear:
    numberCount = {}

    # Track the most frequent number and how often it appears:
    mostFreqNumber = None
    mostFreqNumberCount = 0

    # Loop through all the numbers, counting how often they appear:
    for number in numbers:
        # If the number hasn't appeared before, set it's count to 0.
        if number not in numberCount:
            numberCount[number] = 0
        # Increment the number's count:
        numberCount[number] += 1
        # If this is more frequent than the most frequent number, it
        # becomes the new most frequent number:
        if numberCount[number] > mostFreqNumberCount:
            mostFreqNumber = number
            mostFreqNumberCount = numberCount[number]
    # The function returns the most frequent number:
    return mostFreqNumber
```

Exercise #17: Dice Roll

```
# Import the random module for its randint() function.
import random

def rollDice(numberOfDice):
    # Start the sum total at 0:
    total = 0
    # Run a loop for each die that needs to be rolled:
    for i in range(numberOfDice):
        # Add the amount from one 6-sided dice roll to the total:
        total += random.randint(1, 6)
    # Return the dice roll total:
    return total
```

Exercise #18: Buy 8 Get 1 Free

```
def getCostOfCoffee(numberOfCoffees, pricePerCoffee):
    # Track the total price:
    totalPrice = 0
    # Track how many coffees we have until we get a free one:
    cupsUntilFreeCoffee = 8

    # Loop until the number of coffees to buy reaches 0:
    while numberOfCoffees > 0:
        # Decrement the number of coffees left to buy:
        numberOfCoffees -= 1

        # If this cup of coffee is free, reset the number to buy until
        # a free cup back to 8:
        if cupsUntilFreeCoffee == 0:
```

```
            cupsUntilFreeCoffee = 8
        # Otherwise, pay for a cup of coffee:
        else:
            # Increase the total price:
            totalPrice += pricePerCoffee
            # Decrement the coffees left until we get a free coffee:
            cupsUntilFreeCoffee -= 1

    # Return the total price:
    return totalPrice
```

Alternate Solution:

```
def getCostOfCoffee(numberOfCoffees, pricePerCoffee):
    # Calculate the number of free coffees we get in this order:
    numberOfFreeCoffees = numberOfCoffees // 9

    # Calculate the number of coffees we will have to pay for in this order:
    numberOfPaidCoffees = numberOfCoffees - numberOfFreeCoffees

    # Calculate and return the price:
    return numberOfPaidCoffees * pricePerCoffee
```

Exercise #19: Password Generator

```
# Import the random module for its randint() function.
import random

# Create string constants that for each type of character:
LOWER_LETTERS = 'abcdefghijklmnopqrstuvwxyz'
UPPER_LETTERS = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
NUMBERS = '1234567890'
SPECIAL = '~!@#$%^&*()_+'

# Create a string with all of these strings combined:
ALL_CHARS = LOWER_LETTERS + UPPER_LETTERS + NUMBERS + SPECIAL

def generatePassword(length):
    # 12 is the minimum length for passwords:
    if length < 12:
        length = 12

    # Create a password variable that starts as an empty list:
    password = []
    # Add a random character from the lowercase, uppercase, digits, and
    # punctuation character strings:
    password.append(LOWER_LETTERS[random.randint(0, 25)])
    password.append(UPPER_LETTERS[random.randint(0, 25)])
    password.append(NUMBERS[random.randint(0, 9)])
    password.append(SPECIAL[random.randint(0, 12)])

    # Keep adding random characters from the combined string until the
    # password meets the length:
    while len(password) < length:
        password.append(ALL_CHARS[random.randint(0, 74)])

    # Randomly shuffle the password list:
```

```
    random.shuffle(password)

    # Join all the strings in the password list into one string to return:
    return ''.join(password)
```

Exercise #20: Leap Year

```
def isLeapYear(year):
    # Years divisible by 400 are leap years:
    if year % 400 == 0:
        return True
    # Otherwise, years divisible by 100 are not leap years:
    elif year % 100 == 0:
        return False
    # Otherwise, years divisible by 4 are leap years:
    elif year % 4 == 0:
        return True
    # Otherwise, every other year is not a leap year:
    else:
        return False
```

Exercise #21: Validate Date

```
# Import the leapyear module for its isLeapYear() function:
import leapyear

def isValidDate(year, month, day):
    # If month is outside the bounds of 1 to 12, return False:
    if not (1 <= month <= 12):
        return False

    # If the year is a leap year and the date is Feb 29th, it is valid:
    if leapyear.isLeapYear(year) and month == 2 and day == 29:
        return True

    # Check for invalid dates in 31-day months:
    if month in (1, 3, 5, 7, 8, 10, 12) and not (1 <= day <= 31):
        return False
    # Check for invalid dates in 30-day months:
    elif month in (4, 6, 9, 11) and not (1 <= day <= 30):
        return False
    # Check for invalid dates in February:
    elif month == 2 and not (1 <= day <= 28):
        return False

    # Date passes all checks and is valid, so return True:
    return True
```

Exercise #22: Rock, Paper, Scissors

```
def rpsWinner(move1, move2):
    # Check all six possible combinations with a winner and return it:
    if move1 == 'rock' and move2 == 'paper':
        return 'player two'
```

```
    elif move1 == 'rock' and move2 == 'scissors':
        return 'player one'
    elif move1 == 'paper' and move2 == 'scissors':
        return 'player two'
    elif move1 == 'paper' and move2 == 'rock':
        return 'player one'
    elif move1 == 'scissors' and move2 == 'rock':
        return 'player two'
    elif move1 == 'scissors' and  move2 == 'paper':
        return 'player one'
    # For all other combinations, it is a tie:
    else:
        return 'tie'
```

Exercise #23: 99 Bottles of Beer

```
# Loop from 99 to 2, displaying the lyrics to each stanza.
for numberOfBottles in range(99, 1, -1):
    print(numberOfBottles, 'bottles of beer on the wall,')
    print(numberOfBottles, 'bottles of beer,')
    print('Take one down,')
    print('Pass it around,')

    # If there is only one, print "bottle" instead of "bottles".
    if (numberOfBottles - 1) == 1:
        print('1 bottle of beer on the wall,')
    else:
        print(numberOfBottles - 1, ' bottles of beer on the wall,')

# The last stanza has singular "bottle" and a different final line:
print('1 bottle of beer on the wall,')
print('1 bottle of beer,')
print('Take one down,')
print('Pass it around,')
print('No more bottles of beer on the wall!')
```

Exercise #24: Every 15 Minutes

```
# Loop over am and pm:
for meridiem in ['am', 'pm']:
    # Loop over every hour:
    for hour in ['12', '1', '2', '3', '4', '5', '6', '7', '8', '9', '10', '11']:
        # Loop over every 15 minutes:
        for minutes in ['00', '15', '30', '45']:
            # Print the time:
            print(hour + ':' + minutes + ' ' + meridiem)
```

Exercise #25: Multiplication Table

```
# Print the heading of each column:
print('  | 1  2  3  4  5  6  7  8  9 10')
print('--+------------------------------')

# Loop over all numbers from 1 to 10:
```

```
for column in range(1, 11):
    # Print the number label on the right side:
    print(str(column).rjust(2) + '|', end='')

    # Loop over all numbers from 1 to 10:
    for row in range(1, 11):
        # Print the product, padded to two digits, followed by a space:
        print(str(column * row).rjust(2) + ' ', end='')
    # After the loop, print a newline to end the row:
    print()
```

Exercise #26: Handshakes

```
def printHandshakes(people):
    # The total number of handshakes starts at 0:
    numberOfHandshakes = 0
    # Loop over every index in the people list except the last:
    for i in range(len(people) - 1):
        # Loop over every index in the people list after index i:
        for j in range(i + 1, len(people)):
            # Print a handshake between the people at index i and j:
            print(people[i], 'shakes hands with', people[j])
            # Increment the total number of handshakes:
            numberOfHandshakes += 1
    # Return the total number of handshakes:
    return numberOfHandshakes
```

Exercise #27: Rectangle Drawing

```
def drawRectangle(width, height):
    # Special case: If width or height is less than 1, draw nothing:
    if width < 1 or height < 1:
        return

    # Loop over each row:
    for row in range(height):
        # Loop over each column in this row:
        for column in range(width):
            # Print a hashtag:
            print('#', end='')
        # At the end of the row, print a newline:
        print()
```

Exercise #28: Border Drawing

```
def drawBorder(width, height):
    # Special case: If the width or height is less than two, draw nothing:
    if width < 2 or height < 2:
        return

    # Print the top row:
    print('+' + ('-' * (width - 2)) + '+')

    # Loop for each row (except the top and bottom):
```

```
    for i in range(height - 2):
        # Print the sides:
        print('|' + (' ' * (width - 2)) + '|')

    # Print the bottom row:
    print('+' + ('-' * (width - 2)) + '+')
```

Exercise #29: Pyramid Drawing

```
def drawPyramid(height):
    # Loop over each row from 0 up to height:
    for rowNumber in range(height):
        # Create a string of spaces for the left side of the pyramid:
        leftSideSpaces = ' ' * (height - (rowNumber + 1))
        # Create the string of hashtags for this row of the pyramid:
        pyramidRow = '#' * (rowNumber * 2 + 1)
        # Print the left side spaces and the row of the pyramid:
        print(leftSideSpaces + pyramidRow)
```

Exercise #30: 3D Box Drawing

```
def drawBox(size):
    # Special case: Draw nothing if size is less than 1:
    if size < 1:
        return

    # Draw back line on top surface:
    print(' ' * (size + 1) + '+' + '-' * (size * 2) + '+')

    # Draw top surface:
    for i in range(size):
        print(' ' * (size - i) + '/' + ' ' * (size * 2) + '/' + ' ' * i + '|')

    # Draw top line on top surface:
    print('+' + '-' * (size * 2) + '+' + ' ' * size + '+')

    # Draw front surface:
    for i in range(size - 1, -1, -1):
        print('|' + ' ' * (size * 2) + '|' + ' ' * i + '/')

    # Draw bottom lie on front surface:
    print('+' + '-' * (size * 2) + '+')

# In a loop, call drawBox() with arguments 1 to 5:
for i in range(1, 6):
    drawBox(i)
```

Exercise #31: Convert Integers to Strings

```
def convertIntToStr(integerNum):
    # Special case: Check if integerNum is 0, and return '0' if so:
    if integerNum == 0:
        return '0'
```

```
    # This dictionary maps single integer digits to string digits:
    DIGITS_INT_TO_STR = {0: '0', 1: '1', 2: '2', 3: '3', 4: '4',
        5: '5', 6: '6', 7: '7', 8: '8', 9: '9'}

    # Make a note whether the number is negative or not, and make
    # integerNum positive for the rest of the function's code:
    if integerNum < 0:
        isNegative = True
        integerNum = -integerNum
    else:
        isNegative = False

    # stringNum holds the converted string, and starts off blank:
    stringNum = ''

    # Keeping looping while integerNum is greater than zero:
    while integerNum > 0:
        # Mod the integerNum by 10 to get the digit in the ones place:
        onesPlaceDigit = integerNum % 10
        # Put the corresponding string digit at the front of stringNum:
        stringNum = DIGITS_INT_TO_STR[onesPlaceDigit] + stringNum
        # Divide integerNum by ten to remove one entire digit place:
        integerNum //= 10

    # If the number was originally negative, add a minus sign:
    if isNegative:
        return '-' + stringNum
    else:
        return stringNum
```

Exercise #32: Convert Strings to Integers

```
def convertStrToInt(stringNum):
    # This dictionary maps string digits to single integer digits:
    DIGITS_STR_TO_INT = {'0': 0, '1': 1, '2': 2, '3': 3, '4': 4,
        '5': 5, '6': 6, '7': 7, '8': 8, '9': 9}

    # Make a note whether the number is negative or not, and make
    # integerNum positive for the rest of the function's code:
    if stringNum[0] == '-':
        isNegative = True
        stringNum = stringNum[1:]
    else:
        isNegative = False

    # integerNum holds the converted integer, and starts off at 0:
    integerNum = 0

    # Loop over the digits in the string from left to right:
    for i in range(len(stringNum)):
        # Get the integer digit from the string digit:
        digit = DIGITS_STR_TO_INT[stringNum[i]]
        # Add this to the integer number:
        integerNum = (integerNum * 10) + digit

    # If the number was originally negative, make the integer
```

```
    # negative before returning it:
    if isNegative:
        return -integerNum
    else:
        return integerNum
```

Exercise #33: Comma-Formatted Numbers

```
def commaFormat(number):
    # Convert the number to a string:
    number = str(number)

    # Remember the fractional part and remove it from the number, if any:
    if '.' in number:
        fractionalPart = number[number.index('.'):]
        number = number[:number.index('.')]
    else:
        fractionalPart = ''

    # Create a variable to hold triplets of digits and the
    # comma-formatted string as it is built:
    triplet = ''
    commaNumber = ''

    # Loop over the digits starting on the right side and going left:
    for i in range(len(number) - 1, -1, -1):
        # Add the digits to the triplet variable:
        triplet = number[i] + triplet
        # When the triplet variable has three digits, add it with a
        # comma to the comma-formatted string:
        if len(triplet) == 3:
            commaNumber = triplet + ',' + commaNumber
            # Reset the triplet variable back to a blank string:
            triplet = ''

    # If the triplet has any digits left over, add it with a comma
    # to the comma-formatted string:
    if triplet != '':
        commaNumber = triplet + ',' + commaNumber

    # Return the comma-formatted string:
    return commaNumber[:-1] + fractionalPart
```

Exercise #34: Uppercase Letters

```
# Map the lowercase letters to uppercase letters.
LOWER_TO_UPPER = {'a': 'A', 'b': 'B', 'c': 'C', 'd': 'D', 'e': 'E', 'f': 'F', 'g':
'G', 'h': 'H', 'i': 'I', 'j': 'J', 'k': 'K', 'l': 'L', 'm': 'M', 'n': 'N', 'o': 'O',
'p': 'P', 'q': 'Q', 'r': 'R', 's': 'S', 't': 'T', 'u': 'U', 'v': 'V', 'w': 'W', 'x':
'X', 'y': 'Y', 'z': 'Z'}

def getUppercase(text):
    # Create a new variable that starts as a blank string and will
    # hold the uppercase form of text:
    uppercaseText = ''
```

```
    # Loop over all the characters in text, adding non-lowercase
    # characters to our new string:
    for character in text:
        if character in LOWER_TO_UPPER:
            # Append the uppercase form to the new string:
            uppercaseText += LOWER_TO_UPPER[character]
        else:
            uppercaseText += character

    # Return the uppercase string:
    return uppercaseText
```

Exercise #35: Title Case

```
def getTitleCase(text):
    # Create a titledText variable to store the titlecase text:
    titledText = ''
    # Loop over every index in text:
    for i in range(len(text)):
        # The character at the start of text should be uppercase:
        if i == 0:
            titledText += text[i].upper()
        # If the character is a letter and the previous character is
        # not a letter, make it uppercase:
        elif text[i].isalpha() and not text[i - 1].isalpha():
            titledText += text[i].upper()
        # Otherwise, make it lowercase:
        else:
            titledText += text[i].lower()
    # Return the titled cased string:
    return titledText
```

Exercise #36: Reverse String

```
def reverseString(text):
    # Convert the text string into a list of character strings:
    text = list(text)
    # Loop over the first half of indexes in the list:
    for i in range(len(text) // 2):
        # Swap the values of i and it's mirror index in the second
        # half of the list:
        mirrorIndex = len(text) - 1 - i
        text[i], text[mirrorIndex] = text[mirrorIndex], text[i]
    # Join the list of strings into a single string and return it:
    return ''.join(text)
```

Exercise #37: Change Maker

```
def makeChange(amount):
    # Create a dictionary to keep track of how many of each coin:
    change = {}

    # If the amount is enough to add quarters, add them:
    if amount >= 25:
```

```
        change['quarters'] = amount // 25
        # Reduce the amount by the value of the quarters added:
        amount = amount % 25
    # If the amount is enough to add dimes, add them:
    if amount >= 10:
        change['dimes'] = amount // 10
        # Reduce the amount by the value of the dimes added:
        amount = amount % 10
    # If the amount is enough to add nickels, add them:
    if amount >= 5:
        change['nickels'] = amount // 5
        # Reduce the amount by the value of the nickels added:
        amount = amount % 5
    # If the amount is enough to add pennies, add them:
    if amount >= 1:
        change['pennies'] = amount

    return change
```

Exercise #38: Random Shuffle

```
# Import the random module for its randint() function.
import random

def shuffle(values):
    # Loop over the range of indexes from 0 up to the length of the list:
    for i in range(len(values)):
        # Randomly pick an index to swap with:
        swapIndex = random.randint(0, len(values) - 1)
        # Swap the values between the two indexes:
        values[i], values[swapIndex] = values[swapIndex], values[i]
```

Exercise #39: Collatz Sequence

```
def collatz(startingNumber):
    # If the starting number is 0 or negative, return an empty list:
    if startingNumber < 1:
        return []

    # Create a list to hold the sequence, beginning with the starting number:
    sequence = [startingNumber]
    num = startingNumber
    # Keep looping until the current number is 1:
    while num != 1:
        # If odd, the next number is 3 times the current number plus 1:
        if num % 2 == 1:
            num = 3 * num + 1
        # If even, the next number is half the current number:
        else:
            num = num // 2
        # Record the number in the sequence list:
        sequence.append(num)

    # Return the sequence of numbers:
    return sequence
```

Exercise #40: Merging Two Sorted Lists

```
def mergeTwoLists(list1, list2):
    # Create an empty list to hold the final sorted results:
    result = []

    # Start i1 and i2 at index 0, the start of list1 and list2:
    i1 = 0
    i2 = 0

    # Keeping moving up i1 and i2 until one reaches the end of its list:
    while i1 < len(list1) and i2 < len(list2):
        # Add the smaller of the two current items to the result:
        if list1[i1] < list2[i2]:
            # Add list1's current item to the result:
            result.append(list1[i1])
            # Increment i1:
            i1 += 1
        else:
            # Add list2's current item to the result:
            result.append(list2[i2])
            # Increment i2:
            i2 += 1

    # If i1 is not at the end of list1, add the remaining items from list1:
    if i1 < len(list1):
        for j in range(i1, len(list1)):
            result.append(list1[j])
    # If i2 is not at the end of list2, add the remaining items from list2:
    if i2 < len(list2):
        for j in range(i2, len(list2)):
            result.append(list2[j])

    # Return the merged, sorted list:
    return result
```

Exercise #41: Rot 13 Encryption

```
def rot13(text):
    # Create an encryptedText variable to store the encrypted string:
    encryptedText = ''
    # Loop over each character in the text:
    for character in text:
        # If the character is not a letter, add it as-is to encryptedText:
        if not character.isalpha():
            encryptedText += character
        # Otherwise calculate the letter's "rotated 13" letter:
        else:
            rotatedLetterOrdinal = ord(character) + 13
            # If adding 13 pushes the letter past Z, subtract 26:
            if character.islower() and rotatedLetterOrdinal > 122:
                rotatedLetterOrdinal -= 26
            if character.isupper() and rotatedLetterOrdinal > 90:
                rotatedLetterOrdinal -= 26

            # Add the encrypted letter to encryptedText:
```

```
            encryptedText += chr(rotatedLetterOrdinal)

    # Return the encrypted text:
    return encryptedText
```

Exercise #42: Bubble Sort

```
def bubbleSort(numbers):
    # The outer loop loops i over all but the last number:
    for i in range(len(numbers) - 1):
        # The inner loop loops j starting at i to the last number:
        for j in range(i, len(numbers)):
            # If the number at i is greater than the number at j, swap them:
            if numbers[i] > numbers[j]:
                numbers[i], numbers[j] = numbers[j], numbers[i]
    # Return the now-sorted list:
    return numbers
```

ABOUT THE AUTHOR

Al Sweigart is a software developer, fellow of the Python Software Foundation, and author of several programming books with No Starch Press, including the worldwide bestseller Automate the Boring Stuff with Python. His last name rhymes with "why dirt." His Creative Commons licensed works are available at *https://www.inventwithpython.com*. His cat Zophie weighs 10 pounds.

al@inventwithpython.com

https://alsweigart.com

https://twitter.com/AlSweigart

https://github.com/asweigart

https://www.youtube.com/user/Albert10110

https://www.patreon.com/AlSweigart

Black Lives Matter

Trans Rights Are Human Rights