# Software Craftsmanship

## *Python Workbook*

David Souther

2020-11-29

# Table of Contents

# Introduction

Welcome to the Software Craftsmanship Python examples. These examples will lead you through a variety of programming exercises in conjunction with the main Software Craftsmanship lessons. Programming is a skill, and like any other skill, it takes practice to become truly proficient. The exercises in these examples range from "monkey-see-monkey-do" problems getting you used to typing exactly as a computer expects, to free-ranging ideas on programs you might be interested in writing for yourself.

You will want to complete every "MSMD" section exactly as written. Computers are not forgiving when it comes to what you type in your program, and these exercises are a safe way to begin exploring the world of computer programming. The exercises for each chapter build on the MSMD section, and are opportunities for you to apply what you've learned conceptually to a program. You should make an attempt to complete all the exercises. Finally, projects are larger scale ideas for programs that you would be able to write with the tools presented by that point in the main text. The more you program, the better a programmer you'll be, but you'll probably want to pick and choose projects that interest you.

# python

In this book, we'll be using Python 3. The easiest way to begin using Python 3 is by using Github Codespaces, which creates a virtual linux computer in the cloud for you to use. It is already configured to develop all of the programs in this book. By using codespaces, you will be up and running immediately, with no need to install any programs yourself. However, codespaces are not free. The are billed per hour of activity, though they do deactivate after 30 minutes of inactivity. At the time of this writing, the basic tier of Codespaces are more than sufficient for anything you will do in this book, and costs $0.085 per hour.

| **NOTE** | Codespaces is BETA, and requires sign up. Update this paragraph when it's public. |
|---|---|

If you are using your personal computer, python is included with many Linux distributions, but if you are using macOS (OSX) or Windows, you will need to install it. On Windows and Mac, you will need to download and run the windows installer or the mac osx installer respectively. If it is not installed by default with your linux distribution, follow the python documentation for getting an up to date copy. After installing Python, instructions in this workbook are exactly the same for local development or using Codespaces. Choose one, and stick with it.

# First Program

## Editor

An editor is a program you'll use to write the source code of your programs. It's different from a word processor in that only the characters you type go into the program file. For instance, a word program will have information about what is a heading, where the tab stops are, and where any tables might be. In contrast, the text editor only saves the actual letters and tab you'd type. Any formatting or coloring you see is added after-the-fact, and is not part of the original source code.

There are numerous code editors available, ranging from free to hundreds of dollars, and from having almost no features to being able to write significant portions of your source code and manage large pieces of your program for you. We are going to choose a free editor with many nice features, but one that still makes you handle most of the programming yourself. Managing your project on your own will help you be a better programmer later.

If you're using the recommended Github Codespaces, the editor is Visual Studio Code (or VSCode for short). If you are wanting to install it on your own computer, you can follow the link. It is well supported, has a tremendously rich feature set and ecosystem of extensions, and best of all, is completely free. When you visit the VSCode page, it should suggest an appropriate download link for your platform.

> **Why can't I just use TextEdit on a mac or Notepad on Windows?** You could, but neither of those programs have any tools to help you program. Most importantly, neight has syntax highlighting. Visual Studio Code will color different parts of your code neatly, allowing you to see at a glance important pieces of your program, and often allows you to catch typing mistakes before you even run the program. It also has numerous language integrations, from running code to suggesting likely completions as you type.

When VSCode is first opened, you will need to make a new folder.

Once you have your project folder, create a file and name it `hello.py`. Type this into the file and save:

*Hello, world in python*

```
print("Hello, world!")
print(2 + 3, 12/5, 152 * 12.6)
```

Use VSCode to execute in the terminal.

## Terminal

Throughout the book, we will be using an editor and a command prompt almost exclusively to write and run our programs. A simple command prompt and text editor offer numerous advantages over other programming setups like Integrated Development Environments (IDEs) and visual programming tools. All the reasons stem from the fact that the command prompt is much closer to the code than any of the other tools allow. While many common operations are streamlined, knowing how to use the command line effectively is crucial to overcoming many of the common problems you will encounter in the course of your programming.

### VSCode

Visual Studio Code has a built-in terminal pane that opens on-demand below your code. It uses the default terminal for your system, so read through the appropriate section below, and remember that you will always have this available right inside VSCode.

### Running the program

In Visual Studio Code, you can quickly and easily run a program in the built-in terminal by right-clicking the file and selecting "Run Python File in Terminal".

And you should see the output

```
Hello, world!
5 2.4 1915.2
```

Congratulations! You've written your first program!

![vscode hello python](./vscode_intro_py.png)

### Oh no! Something went wrong!

#### Selecting a Python Version

When you run a python file, VSCode will open the terminal below your code, and execute the python file using the currently selected python version. VSCode may choose to use python 2, which will cause problems with some programs we run. To fix this, click the "Settings" gear in the bottom-left of VSCode, select "Command Pallete" at the bottom of the list, type "Python: Select Interpreter" in the command list, and choose a Python 3 version.

#### Anything Else

This is a lot of new activity for many readers, and it's ok if something went wrong. I'll go through the most common issues here. Keep this page handy, because even if you don't have any of these problems this time, they may accidentally come up again later.

**Early readers: Please send me a message with any problems you have, so I can include them here.**

## Exercises

1. **Say more things** Make your program say
    a. "Time for Breakfast!"
    b. "Goodnight, world."
    c. "Programming is fun!"
2. **Say even more things!** Make your program say another five things.
3. **Say some math.** Make your program say the results of more mathematical expressions. We'll cover arithmetic more in the next chapter.

When you've done some or all of these, get started in [chapter 1](../../01_basic_types_and_control_flow/README.md)

# Basic Types and Control Flow

## Basic Types

This exercise will give you practice typing programs that use many different… types! (Get it… typing… types?) In the example, there are many instances of using variables that have integers, floating point numbers (floats), characters, and strings. There are also some basic control flow statements at the end. If you haven't read the second half of the chapter, type them in exactly as they're written and think about what you expect the result of running them will be. If you have read the second half of the chapter, well, do the same, but you'll have a hint on what they are expected to do!

Looking back on the steps from the introduction "Hello, World!", start by opening a new python file your text editor, and write these two lines of code:

```
print("Hello, world")
print('Hello, again')
```

Save the file (in a new folder for this chapter) as `types.py`. Open a command prompt, cd to the new folder you created, and type `python types.py`. You should see:

```
Hello, world
Hello, again
```

If you have trouble with these steps, go back to [the first exercise](../../00_introduction/python) and follow the instructions there, but for this program. When you have it running, add these next few lines:

```
print(2, 3, 5, 7, 11)

print(0.5, 1.27, 3.141)

print('h', 'e', 'l', 'l', 'o')
```

Type them at the end of `types.py`, save the file, and run it in your terminal again. (Leaving the terminal open so you don't have to navigate to the directory every time would be a good idea ;)

You should see

```
Hello, world
Hello, again
2 3 5 7 11
0.5 1.27 3.141
h e l l o
```

Great! Are you having fun yet?

Let's go back and look very closely at what each of these is. On every line you've typed to this point, you started with the command `print`. This is a built in piece of Python that takes whatever comes after it and outputs that value. The first two lines you typed, `print("Hello, world")` and `print('Hello, world')`, are very similar. The only difference is that one uses the single quotation mark, and the other uses the double quotation mark. Both the single and double quotation marks wrap some text that you want to declare as a String. Recall from the main text that a string is some arbitrary list of characters. At this point, the single and double quotes are almost interchangeable. You would use one when you want to include the other symbol in your string itself. That said, you should choose a primary one to stick with - I encourage double quotes for Python.

The second three lines each show off a different data type. `print(2 3 5 7 11)` outputs several integers. Integers in Python store whole numbers only. The folowing line, `print(0.5, 1.27, 3.141)`, outputs floating point numbers, or numbers that can be fractional. Doing arithmetic with integers can only make integers, which we'll see in a later block of code. The last line here creates several one-character strings, or just characters. Python treats the two as the same. Strings, Integers, and Floating Point Numbers make up the entirety of data you will ever work with in a computer. From here, it's all about learning new ways to combine them!

The next code block is a bit bigger. I would recommend only doing a section at a time before running the program - the sooner you catch a mistake, the easier it will be to correct!

```
i = 2
j = 3
k = 5

print(i, j, k)

m = i * j
n = j + k

print(m, n)

x = 0.5
y = 1.27
pi = 3.141

print(x, y, pi)
```

When it's running, you should see

```
Hello, world
Hello, again
2 3 5 7 11
0.5 1.27 3.141
h e l l o
2 3 5
6 8
0.5 1.27 3.141
```

Working with numbers as they're typed in isn't very useful. In this block of code, we create several variables. Variables are words in your program that refer to a piece of data. That piece of data we call its value. The value a variable has can change over the course of a program. We change the value of a variable with the `=` sign. It's important to remember that when we use a single equal sign in a program, it acts as a verb, "to make equal" - it makes the variable on the left have the value of whatever is on the right for the rest of the program. When a variable is anywhere else in the program, like in these print lines, the program simply uses the value currently assigned to the variable.

### Variable names

A note on naming: variables can be as long as you want, but there are a few rules. First, they can't have any spaces in them. Second, they can't start with a digit (0-9). But they can have a digit in them, and you can also use `\_`, an underscore, to separate "words" in a long variable. In fact, Pythonistas are encouraged to!

Moving a bit faster, for the biggest section yet:

```
circumference_1 = pi * y
circumference_2 = pi * n

print(circumference_1, circumference_2)

average = (i + j + k) / 3

print(average)

a = 3
b = 8.6
c = 2.12

import math
discriminant = math.sqrt((b * b) - (4 * a * c))
denominator = 2 * a
x1 = (-b + discriminant) / denominator
x2 = (-b - discriminant) / denominator

print(x1, x2)
```

Remember, we're just adding these blocks to the end of the types.py file. When it's correct, the output will be

```
Hello, world
Hello, again
2 3 5 7 11
0.5 1.27 3.141
h e l l o
2 3 5
6 8
0.5 1.27 3.141
3.98907 25.128
3.33333333333
-0.272395015515 -2.59427165115
```

Oh boy! That's a lot of math! Ok, let's break it down. First, there's not much in these few lines except arithmetic - addition, subtraction, multiplication, and division. The first four lines calculate two diffrent circumferences, using the variables and values we typed above for `y`, `n`, and `pi`. The `=` sign says "assign the value from the right to the variable on the left". The value on the right, or "right hand side", is `pi * y` and `pi * n`. `pi` was assigned `3.141` above, and `y` was assigned `1.27`, so this right hand side is `3.141 * 1.27`, which is equal to `3.98907`. That value, `3.98907`, is assigned to `circumference_1`. The same thing happens for `circumference_2`, but with `n`. `n` was assigned the value `3 + 5`, or `8`, which when multiplied by `3.141` is `25.128`, the value stored in `circumference_2`.

Let's take a look at the average line. We add `i`, `j`, and `k`. From above, `i = 2`, `j = 3`, and `k = 5`. When those are all added together, they equal `10`. The parenthesese are there to tell Python to do the addition first, before the division. We then divide the integer 10 by 3, and in python 3 the value of

the arithmetic is ALWAYS a float! So we get the extremely accurate `3.33333333333`, which is not exactly correct but as precise as the computer can calculate. (See the appendix on computer arithmetic if you want the gory, mathy details!)

We've covered addition, multiplication, and division. Subtraction should be easy to figure out. The next section does some math with a square root. Specifically, the next few lines calculate the `x` values that the equation `3x\^2 + 8.6x + 2.12 = 0` is equal to `0`. This equation describes a parabola, which is the mathematical name for the shape a projectile travels in - a thrown basketball, or a cannon ball fired from artillery. (Some of the earliest computers were created and used by the US Navy to calculate firing angles and distances for battleships during World War II.)

There are two values of `x` that the equation equals zero, and using the quadratic formula, we can find the two values. However, the quadratic formula requires we take a square root of a number. There's no key for square root, so instead Python provides a variety of utilities in the `math` package. Similar to `print`, but we need to tell Python explicitly that we want them. To do that, we type the line `import math`. The quadratic formula has two intermediate calculations, which we perform with `discriminant = math.sqrtb * b) - (4 * a * c` and `denominator = 2 * a`. Multiplication has a higher priority than subtraction, so in the discriminant it will happen first, but we add parens to make it clear what the order is.

Finally, we use those two intermediate variable values to calculate the two `x` values where `3x\^2 + 8.6x + 2.12 = 0` holds true.

Whew! That's a lot of math! But the computer does it all, correctly, every time. That alone should save your skin in a math class or two!

Let's skip ahead of math for awhile, and look at those strings again. We're going to cover what, exactly, this notation means in the next chapter, but for now practice the typing and think about what the output means the code is doing.

```
a = "Hello, world"
b = 'Hello, world'
c = "This is" + " more text"

print(a, b, c)

print(len(a), len(b), len(c))
```

The output of the program at this point is

```
Hello, world
Hello, again
2 3 5 7 11
0.5 1.27 3.141
h e l l o
2 3 5
6 8
0.5 1.27 3.141
3.99542 25.168
3 3.33333333333
-0.272395015515 -2.59427165115
Hello, world Hello, world This is more text
12 12 17
```

The first two lines should make sense - a variable can have a value that is a string, as well as an integer or float. The third line shows something new. We can add strings together? Yes! Technically, the operation is called "concatenation", but it uses the `+` character, and simply joins the two strings into one single string, side by side. It doesn't add a space or anything, you have to do that on your own. You can get the integer length, or number of characters in a string, using the `len(string_variable)` command.

One last Monkey-see-monkey-do exercise, practicing typing full statements rather than simple expressions:

```
MIN_BALANCE = 25
current_balance = 30
transaction_amount = 10

if (current_balance - transaction_amount) < MIN_BALANCE:
    print("This transaction is too large.")
else:
    current_balance = current_balance - transaction_amount

print("Your current balance is: $" + str(current_balance))

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

And the complete output from the first big program file is

```
Hello, world
Hello, again
2 3 5 7 11
0.5 1.27 3.141
h e l l o
2 3 5
6 8
0.5 1.27 3.141
3.99542 25.168
3 3.33333333333
-0.272395015515 -2.59427165115
Hello, world Hello, world This is more text
12 12 17
This transaction is too large.
Your current balance is: $30
0
1
2
3
4
5
6
7
8
9
```

The last two lines give a way to do the same code many times with a single value changing - in this case, printing the variable with the value 0, then 1, then 2, and so forth.

Before that, we have a possible bank application. This bank requires that a savings account must always have at least $25. When someone tries to do a transaction, the program first checks if the account has enough money. If not, it prints a warning to the user. Otherwise, it deducts the transaction amount from the account. Either way, it always tells the user how much money is remaining in the account. We'll cover these more in the next section.

## Exercises

You should do each of these exercises in their own files. They include a lot of math. Don't let the math scare you! We're just using python as a calculator at this point. Do each computation, store it in a variable, and then print that variable out. The goal here is to practice typing python code and running python problems. It is **not** a math course! Formulas for all the exercises are included.

Do as many or as few of the exercises as is interesting to you. This is a chance to play around with a new hobby, and you might find some of these types of calculations are interesting. But it's really not critical to understand all the details - just do what's fun, and then go on to the control flow section!

1. **Pricing rugs** A company makes rugs, and has asked you to calculate the price for their rugs. They have square rugs (`area = length \* length`), rectangular rugs (`area = length * width`), and circular rugs (`area = pi \* radius \* radius`). Rugs are $5 per square foot of finished rug. Write

a program that prints the cost of rugs for these sizes:

a. Square, 1 foot on a side. ($5)

b. Square, 2.5 feet on a side. ($31.25)

c. Rectangular, 3 feet by 5 feet. ($75)

d. Circular, 1.5 foot radius (3 feet across). ($35.33645, using pi = 3.141)

2. **Advanced rugs** The rug company loves your code! They want you to add another feature! They now offer edges on their rugs. Edging costs $1/foot of edge. (The perimeter of a square is length * 4. A rectangle is length * 2 \+ width * 2. A circle's perimeter is Pi * radius * 2.) The same rugs, with edging, should be

   a. $9

   b. $41.25

   c. $91

   d. $44.77

3. **Harder Math** You might recognize the math section in the middle of the program (where we use the 'discriminant' variable) as the quadratic equation. THe quadratic equation is a formula mathemeticians use to determine where a parabola has the value zero, and that physicists use to calculate where a basketball will land when thrown with a certain force.

   The full quadratic equation is, again:

   ```
   import math

   discriminant = math.sqrt((b * b) - (4 * a * c))
   denominator = 2 * a
   x1 = (-b + discriminant) / denominator
   x2 = (-b - discriminant) / denominator
   ```

   the different being a + in the first and a - in the second.

   Solve the quadratic equation for:

   a. A = -2, B = 5, C = 3. http://www.wolframalpha.com/input/?i=-2+**x%5E+2+%2B+5**+x%2B+3[x1 = -0.5, x2 = 3]

   b. A = 1.5, B = -6, C = 4.25. http://www.wolframalpha.com/input/?i=1.5+**x%5E+2+-6**+x%2B+4.25[x1 = 0.919877, x2 = 3.08012]

   c. A = 1, B = 200, C = -0.000015. http://www.wolframalpha.com/input/?i=x+%5E+2+%2B+200+x+-+0.000015 (x1 = 7.5*10^-8, x2 = -200)

4. Click the answer links to see the problem breakdown in Wolfram Alpha.

   Notice in the last the result from Wolfram Alpha is different than the result we got in Python. Formally, the variation of the quadratic formula we use here is "numerically unstable" - when used on certain values, the solution we chose is not capable of giving the right answer. While this might not look like more than a peculiarity today, it is a serious issue in many branches of

software engineering.

Rerun the last calculation using the following formula (7):

```
basis = -b - discriminant
x1 = basis / (2 * a)
x2 = (2 * c) / basis
```

Notice how it is closer to the exact values provided by Wolfram Alpha, but still slightly incorrect.

In this exercise, the details of the math aren't particularly important. The important piece is recognizing there are situations where the first, obvious solution to a programming problem might have subtle errors. It's critical to always check your assumptions and work against some external source. For an interface, grab a friend to test it out. For an algorithm, verify the result against some other program that performs correctly (even if that means by hand). We will come back to this concept of using an external source to test your programs many times throughout the text.

5. **Your Own Maths** I promise, the only reason we're doing this much math is because we don't know how do to anything more interesting until the next chapter! We will start branching out! For now, though, the practice of typing various maths is excellent coding practice.

   In this exercise, choose one or more of these formulas and implement it, using several sets of numbers. For more practice, do more of these!

   a. **Recipe Measurements.** In a recipe book, recipes are given in servings prepared and quantity of ingredients added. An omelette recipe takes 3 eggs and makes 2 servings. A waffle recipe takes 2 cups of flour, and makes 6 waffles. If we wanted to double the recipe, we could multiply the eggs and flour by 2. This would let us make 4 omelettes and 12 waffles, but would require 8 eggs and 4 cups of flour. Or if you were only feeding yourself, you'd halve the recipes, using 1.5 eggs and 1 cup of flour for a half-sized omelette and 3 waffles.

      As an exercise, pull a recipe book off your shelf (or find a recipe online). Use Python to calculate the amount of measurement for doubling, tripling, or halving each recipe.

      > Tip: If you have two variables, `desired_servings` and `recipe_servings`, you can change the scale of the recipe by multiplying every measurement by `desired_servings / recipe_servings`.

   b. **Pieces of trim** When finishing a room, it's common to install trim at the base of the wall. Trim commonly comes in 8' segments. Suppose you have a room that's 12' by 18', with a 3' wide doorway. To cover the base of all the walls, you'll need to cover 57' of wall with trim (12 + 12 + 18 + 18 - 3). This requires 8 pieces of trim - because you can't buy less than a whole piece of trim! Use python to determine the number of pieces of trim you need for rooms of these sizes:

      i. 12' by 18', with one 3' doorway.

ii. 10' by 20', with two 3' doorways.

c. Simplified Division Algorithm When you divide two numbers, there are many ways to express the result. So far, we've been using the floating point format, with some number of decimals after the number. That leads to some odd answers, like with the board-foot calculation being 233.33333333333334. In ancient Greece, Euclid showed how to determine the "remainder" of a division - in this case, remainder 1 when dividing by 3. In Python, there is an operator called "modulus" which returns the remainder of dividing two numbers. It's typed as `%`, as in `3 % 2`, which would be `1`. Try putting `print(5 % 3)` in your python program, and see that it prints 2.

Use the modulus operator to improve your board-foot calculations. Come up with new board-foot calculations that need remainders to be sensible.

a. [Standard Deviation](https://en.wikipedia.org/wiki/Standard_deviation=Basic_examples)
   - Calculate the standard deviation for the sample data - `2, 4, 4, 4, 5, 7, 9 = 5`
   - Given the height of an adult male in inches, calculate which deviation he is in (mean = 70, deviation = 3). Try using the modulus operator.
     - `71 = 1/3`
     - `68 = - 2/3`
     - `78 = 2 2/3`

b. **Projectile Distance** One of the first applied uses of computing machines was in naval artillery situations. Write some code to calculate the range of a projectile fired at various velocities and angles (in radians) from ground level. To calculate sine, use `math.sin(x)`. Assume velocity is in meters per second, so `g = 9.81` in the simplified formula on the wikipedia page.
   - `v = 100, theta = 0.1745; D = 348.581308308` (theta is \~10 degrees)
   - `v = 1000, theta = 0.1745; D = 34858.1308308`
   - `v = 250, theta = 0.6981; D = 6274.18922949` (theta is \~40 degrees)

Congratulations! You have a lot of experience typing Python programs! Getting used to typing and running programs is a skill in itself, independant of the actual programming you will do. This typeing programs is usually called coding, seperate from actually figuring out what a program should do, which is the programming.

# Control Flow

These exercises put the control flow concepts to use. They use the techniques from part 1, types, extensively. Many control flow constructs derive their operation from the types and operations within them, so having condidence from the first half of the chapter is critical to success, really in the rest of your programming career.

As in the first two groups of examples, start by opening a new file in your editor. Type in these two lines.

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

Save the new file, as `control.py`

Run the file - `python3 control.py`

```
0
1
2
3
4
5
6
7
8
9
```

What happens here? This is a "for" loop, which runs the statement `print(i)` 10 times, each time with the variable `i` taking a different value. `print(i)` itself is an expression, that becomes a statement when the start of the new line tells python to end the expression. The `for i ⋯ :` tells python what variable to change within the block. The `range(0, 10)` tells python to use that range for the values of `i`, starting at 0 and going up to, but not inclding, 10. The block is everything after that line, *that is at the next indentation level*. An indentation level is anything with the same number of spaces at the start of the line. `for i in range(0, 10):` has no spaces. ` print(i)` has four spaces. Indentatation level is very important in Python, and we will revist it several times.

## Summations

With that in mind, rewrite the file so it contains this.

```
sum = 0
for i in range(1, 11):
    print(i)
    sum = sum + i
print(sum)
```

Save it, run it, and the results:

```
1
2
3
4
5
6
7
8
9
10
55
```

Take a moment, and think about what happened.

The program started with an additional declaration, setting `sum` to 0. There is an additional line in the for statement, `sum = sum + i`. It is at the same indentation level as `print(i)`, and so runs in the same segment of the program. Finally, `print(sum)` is back at no indentation, so runs after the for statement finishes.

Change this program again:

```
sum = 0
N = input("Summing 1 to N: ")
for i in range(1, (int(N) + 1)):
    sum = sum + i
print(sum)
```

Now when it runs, it will stop and ask you to input a number for N. Choosing 10 will print 55. What does choosing 100 print?

```
Summing 1 to N: 100
5050
```

The `input("Summing 1 to N: ")` is a python way to ask the user to enter a number. Whatever the user types, up to the enter, is the result of the expression. Because what the user types is a string, you need to tell python to treat it as an integer with `int(N)`.

Now that we have a program that sums a few numbers, change it to ask the user to choose between summing or making a product.

```
add = input("Sum or Product? (S/P): ")
if add == "S" or add == "s":
    sum = 0
    N = input("Summing 1 to N: ")
    for i in range(1, (int(N) + 1)):
        sum = sum + i
    print(sum)
elif add == "P" or add == "p":
    prod = 1
    N = input("Product 1 to N: ")
    for i in range(1, (int(N) + 1)):
        prod = prod * i
    print(prod)
else:
    print("Unsupported Operation")
```

As we go along, the programs get bigger! The sum program is still there, lines 3 through 7. They're repeated again, lines 9 through 13. The two parts of the program are separated by an if clause, with three branches. The first `if` is the same as from the text. The second, the `elif`, is a way to do a second check in the program. The `else` handles any time the input isn't `s`, `S`, `p`, `P`. Notice in the `if` conditions, the `or` checks for the two different possible characters. Remember that upper case and lower case characters are different, so checking for both is the polite thing for users.

There is one very important thing to note about the `if` statements - the use of `==` or double equals. To this point, we've been using `=` to be assignment, making a variable have some value. This `==` is a different operation entirely, and is actually comparing two values. Be careful not to confuse them!

Use this program to compute the sum of 1 to 57, and the product of 1 to 100. Python internally handles integers to arbitrary length, making it ideal for many numerical programs. When the number is smaller than some internal integer size (at minimum 32 bits, possibly 64 bits on modern computers), Python will use that for speed. When a calculation needs more precision, it moves to a "long integer" format that allows integers of any size. That said, it is still constrained by things like its memory and processing power - you can't reasonably iterate over more than about 2\^16.

## Roman Numerals

We're going to write a completely new program, so start a new file. Name it `roman.py`, in the same folder as `control.py`. We are going to build this new program up bit by bit, discussing each piece we add.

Start with these lines:

```
print("Decimal to Roman Numeral")
print()
number = int(input("Decimal integer: "))
print()
print("Converting " + str(number) + " to Roman.")
```

This form will become fairly common - print some header content, ask the user for input, tell the user about the input. There is the slight technicality of going between string and number - the `int()` and `str()` parts of line 3 and 5. They will make more sense next chapter, but for now remember that the number `15` and the word with the letters "15" are different in Python's mind.

Running it does what you'd expect:

```
Decimal to Roman Numeral

Decimal integer: 18

Converting 18 to Roman.
```

With the basics out of the way, let's start work on Roman numerals. The rules for Roman Numerals involve making groups of five, starting with "I" characters. A group of five "I"s is replaced with a "V". Two "V"s become an "X". Characters are added from left to right. With this simplified Roman Numeral ruleset, add this to the program:

```
numeral = ""
while number > 0:
    if number >= 10:
        numeral = numeral + "X"
        number = number - 10
    elif number >= 5:
        numeral = numeral + "V"
        number = number - 5
    else:
        numeral = numeral + "I"
        number = number - 1

print(numeral)
```

Running the program, again converting 18 should print out:

```
Decimal to Roman Numeral

Decimal integer: 18

Converting 18 to Roman.
XVIII
```

Huzzah!

Let's take a closer look at what all we just did. We start with `numeral = ""`, which declares and initializes the variable numeral to the empty string. We'll use this variable to build up the roman numeral we finally print out. After that, we start our loop with `while number > 0:`. This implies that we're going to be changing the `number` variable, and ending when it becomes equal to or less than

zero. So we'll probably be subtracting from it!

The body of the loop is what gets repeated. It's what comes after the `:` colon and is at the next indentation level. That's a fancy way of saying that the lines which come afer the `while` statement with the same number of spaces ahead of them! This approach, where leading whitespace is critical to the syntax and semantics of the program, is unique to Python. We'll revisit it several times throughout the book, but for now suffice to say "Be consistent". If you use VSCode, its default is four spaces, and if you hit tab, it will replace that with the requisite four spaces.

Within the `while` loop we have an `if ⋯ elif ⋯ else` control structure. Each `if` branch has a condition. In this case, the conditions are `number >= 10` and `number >= 5`. Python will evaluate them in order, execute the first block whose condition is true, and then skip to after the end of all the `if ⋯ elif ⋯ else` blocks. This sequential ordering is important - it will always peform the first matching case, not the best matching case. If none of the conditions match, it will do the `else` body.

Each body then has a pretty similar setup, like for the 10 case:

```
numeral = numeral + "X"
numer = number - 10
```

This appends (adds it to the end) the next letter of the numeral, and then changes the number we're currently building up. This behavior, where we change some variable of interest and then the variable which controls the loop, is a very common pattern.

Once the associated block of the `if ⋯ elif ⋯ else` finishes, the entire section is complete. Because nothing comes after it in the code at the same indentation level, the program loops back up to the `while number > 0:` line. It evaluates that again, and if it's still true, does the entire body over again! If instead it's false, it goes down to the next line of code (or the end of the file) at the same indent level. In this case, that is `print(numeral)` showing us the value we build up! And being the last line in the file, the progam ends.

#### Wider Range of Numerals

Of course, our program doesn't handle numbers larger than 49, and it doesn't follow the rule that 4 should be "IV", instead of "IIII". Let's fix the second one together. Edit the body of the algorithm (the while loop) to have a new case for when number is 4:

```
    elif (number == 4):
        numeral = numeral + "IV"
        number = 0
```

and run it, using 14:

```
Decimal to Roman Numeral

Decimal integer: 14

Converting 14 to Roman.
XIV
```

It's also a good idea, whenever changing a program, to run it using the old values, to make sure it's still correct.

## Exercises

1. **Roman Numerals** Finish the Roman Numeral program.
   a. The full order of Roman numerals was:
      - I - One
      - V - Five
      - X - Ten
      - L - Fifty
      - C - Hundred
      - D - Five hundred
      - M - One Thousand
   b. The used subractions were
      - IV - Four
      - IX - Nine
      - XL - Forty
      - XC - Ninety
      - CD - Four Hundred
      - CM - Nine Hundred
   c. Sample numbers
      i. 551 - DLI
      ii. 707 - DCCVII
      iii. 90 - DCCCXC
      iv. 1800 - MDCCC
2. **Recipes**
   - Create a new program `measurements.py`. Ask the user how many people they want to serve. Just choose a recipe and hard-code the measurements for the recipe. Scale the ingredients based on how many people the user has entered. After the first recipe, add a few more and let the user choose between several recipes. Make sure your recipes don't all use the same ingredients!

> This will feel like a huge amount of duplication. That is the point. We do not have the tools we need to make this work without duplicating this code. The exercises in this chapter are to get you feeling confident using only the tools we have so far - variables, values, math, and `if` conditions. In chapter 2, we will use **functions** to shock you with how much shorter it can be!

- Some of the recipes you scaled down in the last section might not make a huge amount of sense. Trying to get one and a half eggs? That's a pain! Add conditional statements to your calculations in the earlier exercises that tell the user they can't scale certain recipes if that would result in measurements that are difficult to scale (like fractional eggs).
- While scaling some of these measurements, you may have noticed some odd or difficult measurements. How do you handle 0.125 cups of an ingredient? In a real kitchen, you would switch from 1/8th a cup to using 2 tablespoons! Modify your program to convert between certain unit sizes. You can find tables of measurements conversions online, but most conversions will work by using the conversions of 1 cup is 16 tabelespoons, and 1 tablespoon is 3 teaspoons.

3. **Rugs**

   a. Create a new program, `square_rug.py`. When it runs, ask the user the size of the rug, and whether they want fringe. Print out the cost.

   b. Create two more rugs programs, `rectangular_rug.py` and `circular_rug.py`. Ask the user the appropriate questions and print the cost.

   c. Create a program that combines these three - first ask the user which type of rug they are creating, and then have it choose between those three implementations. Copy/paste the implementations from the earlier parts of the exercise.

These control statements - `while`, `for`, and `if`, are the bread and butter of making programs do interesting things. Many times, we want to do the same calculations on slightly different numbers - `for` and `while`. Other times, we need to do different things based on some condition. In programming, these are called iteration and banching, respectively.

Before we move on to writing our first big program, we're going to step back to the theory and look at [tracing program execution](./TRACING.md).

# HiLo

HiLo is a simple guessing game. The computer chooses a random number, and the player is scored on how quickly they guess the number.

## Type this Program

In a new file, type this program, save it as `hilo.py`, and run it in your terminal!

```
import random
```

```
print("HiLo")
print()
print("This is the game of HiLo.")
print()
print("You will have 6 tries to guess the amount of money in the")
print("HiLo jackpot, which is between 1 and 100 jellybeans. If you")
print("guess the amount, you win $10 for every guess you don't take!")
print("Then you get another chance to win more money. However,")
print("if you do not guess the amount, the game ends!")
print()

winnings = 0
playing = True

while playing:
    guesses = 0
    number = random.randint(1, 10)
    round_finished = False

    while not round_finished:
        guess = input("Your guess: ")

        if guess == "":
            playing = False
            break # Exit the round early

        guess = int(guess)

        if guess == number:
            new_winnings = winnings + ((6 - guesses) * 10)
            print("Got it! You won " + str(new_winnings) + " dollars!")
            winnings = new_winnings
            print("Your total winnings are " + str(winnings) + " dollars!")

            again = input("Play again? (Y/n) ")
            if again != "Y":
                playing = False
            round_finished = True

        else:
            if guess > number:
                print("Your guess was too high!")
            else:
                print("Your guess was too low!")

            guesses = guesses + 1
            if guesses >= 6:
                print("You took too many guesses!")
                playing = False
                round_finished = True
```

```
            print()

print("Goodbye!")
```

Run this a few times, and enjoy playing this little game!

A couple things to call out that we haven't seen before.

We have multiple loops nested, `while playing` and `while not roun_finished`. It's totally fine to do this - when the outer loop gets to the inner loop, the inner loop will run over and over until its condition is no longer true. The program will wrap back around to the outer loop, which will either exit itself or rerun its own body, again running the innter loop. We see the same thing with the nested `if ⋯ else: if⋯ else` (and also, the `elif` and `else` are totally optional!).

There is an extra way to control the loops, rather than relying only on the loop condition. That's the `break` statement. It will immediately end the nearest loop, skipping the rest of the body and also not evaluating the condition. `break` means end the loop right now! There's another command, `continue`, which skips the rest of the loop body but *does* re-check the condition back at the top.

To generate a random number, we use a pair of lines

```
import random
number = random.randint(0, 10)
```

Just like we used `import math` to load a library function for `sqrt`, random gives us access to a bunch of ways to test randomness. The `randint(0, 10)` part picks an integer uniformly between 0 and 10.

## Exercises

1. Assuming you copied the program exactly as presented, there is one bug that should be obvious. Fix the program so it uses the correct range of random numbers.
2. **Terminal Love** You may remember programs in the 80s that used terminal characters for all their screen printing. Before computers were powerful enough to regularly display modern images, programmers would use the extra characters in the ASCII character set to create the precursers to windows, tables, columns, and other graphical interface elements. There were also utilities available to draw those characters in a variety (by which I mean all of eight) different colors. In this series of exercises, we'll make HiLo look like those old games!
    a. **Bold** To start off, we'll include a library (like `random`) for changing the typography of pieces of the program. Create a new file in your directory named `colors.py` and paste the contents of [this file](./colors.py) into it. Save it next to your hilo program. Add `import colors` to your program, below `import random`.

    Edit the first line of the indroduction paragraph to make HiLo bold - `print(colors.BOLD + "HiLo" + colors.NORMAL)`. Notice we have to tell the program to put the printing back to normal, or else all future priting would be in bold. Run the program and see what it does!

    b. **Colors** The colors module has a variety of styles, independent of each other. The typography

can be `BOLD`. Printing can be in `WHITE`, `BLACK`, `RED`, `BLUE`, `GREEN`, `YELLO`, `CYAN`, and `PURPLE`. The background color can be set with `BG_WHITE`, `BG_BLACK`, `BG_RED`, and so on. Play around with the colors, until you like the look and feel of your HiLo game. `NORMAL` puts the terminal back to a normal weight white text on black background.

c. **Start Over** If you've gotten your terminal into a jumbled mess, print `colors.CLS` to clear the screen, and `colors.at(1, 1)` to move the cursor to the top-left corner.

d. **Print at various points** You can move the cursor (where the print happens) using `print(colors.at(r, c))` where r and c are the row and column you want to print at. Row and column start at the top-left of the terminal at `(1, 1)`, and on "standard" terminals there are 80 columns and 24 rows. Note, if you resize the terminal, that will change. When writing this kind of terminal program, it's usually best to stick with those dimentions.

3. **Pauses / Animation** The original Hi Lo program, at the top of the page, doesn't stop between showing the instructions and asking for the first guess. We could stop that with a simple way to ask the user for some input. Try putting `n = raw_input("Press enter key to continue…")` in your program, and you'll see that now the program waits for the user before continuing. In other places, you might not want to wait for the user, but you do want a pause. Using the time module, you can have your program wait a moment.

```
import time

print("Hello, world!")
time.sleep(1.5)
print("A second and a half later...")
```

   - While a pregnant pause in the interface might be interesting, this becomes very useful when putting animation in a program. Try the following HiLo intro scroll:

```
import time

# An 8 frame animation
for i in range(1, 9):
    print(colors.CLS)
    print(colors.at(i, 38), colors.GREEN, "Hi")
    print(colors.at(i+1, 40), colors.RED, "Lo")
    # Half-second delay in each frame
    time.sleep(0.5)

input("Press enter to continue...")
```

   - We'll explore this more in later chapters, but see if you can incorporate this into your program!

4. **Difficulty** Give your program several difficulty levels. You could use easy = 1 to 10, medium = 1 to 100, and hard = 1 to 1000. You could do a custom difficulty. You may want to improve the scoring, so you don't get fewer points when the random number happens to be smaller.

5. **Binary Search** You may have found the "optimal" algorithm is to begin by choosing 50, then

either 25 or 75 depending on if your guess is high or low. This is a strategy called "Binary Search", because there is an either/or decision to look to one side or the other of the numbers you're searching. Rewrite the program so that instead of asking the user for a guess, the program "plays" itself by guessing the number. See how long it takes it to guess any particular number.

If you want to see a full HiLo program, see my [completed example][full_hilo]. It has the animation, a box for input, difficulty levels, and more! Play with your program as much as you like, tweaking colors, moving the box, and whatnot. Once you're happy with your game, you're ready to move on. But first:

Congratulations! You've written your first computer game!

[Wrap Up](../wrap_up.md)

[page437]: http://en.wikipedia.org/wiki/Code_page_437#Interpretation_of_code_points_1.E2.80.9331_and_127 [full_hilo]: https://github.com/DavidSouther/software_craftsmanship/blob/master/01_basic_types_and_control_flow/hilo/hilo.py [box_chars]: http://en.wikipedia.org/wiki/Box-drawing_character

# Functions, Arrays, and Strings

## Functions

Let's start with this program, that defines a function, a reusable block of code, to calculate the price of our square rug from chapter 1:

```
def square_rug_cost(size, fringe):
    area = size ** 2
    cost = area * 5
    if (fringe):
        perimeter = size * 4
        cost = cost + perimiter * 1.5
    return cost

print(square_rug_cost(5, False))
print(square_rug_cost(2.5, True))
```

Type this program as `rug_functions.py` and run it. You should see it print out the two rug costs - 125 and 46.25, respectively.

Let's take a closer look at the anatomy of this function. Recall from the textbook portion that a function needs four things: A name, its arguments (also called parameters), the body, and a return statement. In this `square_rug_cost` function, those look like this:

![diagram of a python function](./digraph_of_a_python_function.png)

In python, a function starts with the key word `def`. That's followed by its name, in this case `square_rug_cost`. It then has two arguments, all inside the parentheses - `(size, fringe)`. Finally, in python, the function declaration (the name and argument list) ends with the `:` (colon). We've seen this in the chapter on control flow as well, and it indicates that we're about to group a bunch of code at the next indentation level.

To get an indentation level, we add a new line (press enter on our keyboard) and indent one level. There are several ways to indent. One is to hit the tab key on your keyboard. Another is to add some number of spaces - at least two, but four and eight are also common. A third way is to let VSCode do it for you! You've probably seen this already when working through the first chapter, that VSCode would automatically add the indendation for you when you started a new line after a `:` colon.

So, why does Python have so many ways to do indentation? Mostly it's historical reasons. For our purposes, we're going to start with using 4 spaces for indentation, because that's VSCode's default, and usually it'll handle it for us. The only rule is to be consistent with the amount of indentation you use in your program.

At the end of the code, we **call** the function twice, once with the values `5` and `False` for its arguments, and then a second time with the values `2.5` and `True`. We print out both values right away. This code is just here to test that the function worked, and we'll replace it with something more substantial.

### One function, many calculations

Now that we have a function that prints a box, what else can we get? Delete the last two lines of file you started (the print lines), and add this code:

```
size = input("Rug size (empty to quit): ")
while size != "":
    wants_fringe = input("Does the rug have fringe (y/n)? ")
    has_fringe = wants_fringe == "y"
    cost = square_rug_cost(float(size), has_fringe)
    print(cost)

    = Get the size for the next rug
    size = input("Rug size (empty to quit): ")
print("Goodbye")
```

We can price (square) rugs over and over!

So now our program just loops as long as the user gives us a size, asks if they want fringe with that, and then calculates and prints the rug price. Instead of providing literal values for the function call, it uses two variables, `size` and `has_fringe`. `has_fringe` is `True` when the user provided `"y"` in their input, and `False` if the user provided any other input. When we call `square_rug_cost`, we look up the values for those variables and provide them to the function. We do **not** provide the variables themselves! Only the values go to the call, and the variables are unrelated!

That's good, but it actually took more lines of code than if we'd just written it all as one block... We'll come back to that. For now, bear with me two more steps. First, we will turn that input into its own function. Then, we're going to do make two similar functions for a rectangular rug. While that will look like more code than we needed, it will pay off when we add circles and start talking about abstraction.

But wait a minute, why do we have `cost` in two different places? Those are actually two completely separate variables, in what we call different scopes. Every function starts a new **scope**, which is a way to set a new group of variables it knows about. A function knows about all the variables in any scope outside of it, as well as all the variables it declares. However, the variables that get declared *inside* the function aren't visible further out. This neatly allows each function to do its own work without stepping on the toes of other functions! Also, the base level of every file is its own scope.

Aside from files, only functions with `def` define new scopes. The body of a `while` loop or `if` block does not create a new scope!

Let's take that while loop, and break it into a function and a loop. Delete everything you typed in the last section, and type this instead.

> If you've gone through appendix 2 on source control, this would be a good time to make a commit, to keep that version of the file which does the loop & calculation in the same block.

```
def compute_square_rug():
    size = float(input("Rug Size: "))
    fringe = input("Does the rug have fringe (y/n)? ")
    fringe = fringe == "y"
    cost = square_rug_cost(size, fringe)
    print(cost)

while input("Price a rug (y/n)? ") == "y":
    compute_square_rug()
```

But why are we adding more code when it just does the same thing!? ARGH! I asked you to stay with me one more step, before we see the pay off. First, delete the last two lines (the while loop that asks whether to price another rug), and then add the following.

Before deleting the loop, this is another good time to make a commit…

```
def rectangular_rug_cost(width, height, fringe):
    area = width * height
    cost = area * 5
    if fringe:
        perimeter = width * 2 + height * 2
        cost += perimeter * 1.5
    return cost

def compute_rectangular_rug():
    width = float(input("Rug width: "))
    height = float(input("Rug height: "))
    fringe = input("Fringe (y/n)?: ") == "y"
    print(rectangular_rug_cost(width, height, fringe))

while input("Price another rug (y/n)? ") == "y":
    rug_type = input("(1) square or (2) rectangular? ")
    if rug_type == "1":
        compute_square_rug()
    elif rug_type == "2":
        compute_rectangular_rug()
```

At this point, we've written four functions. For each rug type, we have two functions - one to perform the calculation, and one to perform the user input. So why would we write all four of these, instead of just one loop? The first answer is to make it easier to read and understand. One big function or loop body would be shorter to write, yes, but in software engineering there's a somewhat paradoxical fact people read any given code many times, while they write it once and edit it maybe a couple more times. You've likely seen this yourself while trying to get your prior programs to work - you would run it, it wouldn't work right or you'd get an error, and then you had to go back through many times reading it to figure out what it was doing.

By separating out each task into functions like this, you are using *abstraction*. Each function lets us move from handling the details of getting input or calculating a rug size, to instead working with the **idea** of getting input or calculating the rug's size. Why is this useful? Mainly, it means that now whenever I think about calculating a rug, I don't need to remember the formulas for its area and perimeter, I only need to know whic arguments are necessary.

It also gives us a good separation of concerns. That's a fancy way of saying the opposite of what was in the last paragraph - from the ouside, I only need to know what arguments to pass the function and it does the work. I can delegate to it, and trust it will do the work. From the inside, I only need to take those arguments and do the calculation. I don't care how I got those arguments! The could come from a command line interface, like here, or in chapter 5, they could come from a web server which provides this functionality to customers online. That separation is a great boon that abstraction using functions gives us.

## Anatomy of a function

Let's cover the critical pieces of a function again.

It starts with the Python keyword `def` - short for "Define", and used here to "define" a function. The next word, `square_rug_cost` or `calculate_rectangular_rug`, is the identifier which tells Python what to name your function. It is the word you'll use to refer to the function in the future, like variables elsewhere. The parentheses mark the **arguments** to the function. Arguments are variables, but variables that are only visible and valid inside the function. The function declaration ends with the `:`.

The definition of the function is everything happening below that line at the next indentation level. Remember that in Python, whitespace is important. An **indentation level** is all code which has the same amount of leading spaces - in our case, four. Just like in `if` and `for` and `while` loops, this code at the same level of indendation is the **function body** is what gets executed each time you call your function.

Ok, let's write one more function, this time to combine the two rugs we had before. Again, delete the "Price another rug" while loop, and add this.

```
def choose_rug():
    type = input("Would you like to price a (S)quare rug or a (R)ectangular rug? ")
    if type == "S" or type == "s":
        compute_square_rug()
    elif type == "R" or type == "r":
        compute_rectangular_rug()

while input("Price another rug (y/N)? ") == "Y":
    choose_rug()
```

What all have we done here? We took a program that had a single flow of data, with variables all over the place, and found small isolated chunks of work. We took those isolted, self-contained bits of work and gave them names by creating functions. We had to create all the functions in our program before the bottom, when we actually call them. Overall we traded some additional global

complexity for less local complexity.

> **Like a recipe** If we were writing a cookbook, we have a lot of different ways to write recipes. We could cram all the steps together, and we could be very explicit about each one. "Take the onions. Remove the skins. Cut the unions into thin slices. Turn on the oven to medium heat. Put a sauce pan on the oven. Heat oil & butter in the pan. When the oil and butter are hot, add the onions. Occaisionally stir the onions for 10 minutes for softened and translucent. For caramelized onions, add sugar at 10 minutes and continue stiring occaisionally for another 35 minutes."
>
> If this were a recipe for a steak dinner, that's a lot of steps that aren't really related to the steak itself! And are we going to repeat those steps every other dish that calls for softened or caramelized onions? I would hope not. Instead we make a new recipe, "Caramelized Onions", and whenever another dish calls for them we just say "2 cups caramelized onions, see page such and such." That's exactly what we're doing with the functions here! Finding self-contained pieces that can be used repeatedly, rather than repeating ourselves every time over and over again.

### Exercises

- **Rugs** Write functions for a circular rug and let users choose circular rugs in the input.
- **Recipe Measurements** Revisit the `measurements.py` file from chapter 1. Create functions for common parts that are duplicated or otherwise should be grouped in the file. No peeking, but [here's my breakdown](./recipe_fns.md)
- Write functions for exercises in chapter 1.

Functions are a really important concept in programming and software engineering, so now, let's go back to the text book and take a close look at how they execute.

## Format strings in Python

Thus far in python, we've been using string concatenation to get the formatting and display we want. In this section, we're going to look at python format string and rewrite the functional hilo we used in the last section using format strings. After this section, the workbook will switch entirely to using format strings as we go forward.

A format string in python looks almost like a normal string, except it has an `f` (for format) before the opening quote, and inside the string, it can reference variables in the surrounding scope by wrapping them using curly braces - `{ }`.

```
balance = 372.15
print(f'Your current balance is ${balance} dollars.')
```

Here we have the variable, `balance`, and the format string, `f'Your current balance is ${balance} dollars.'`. This will format the value of balance into the string and print it out: `Your current balance: $372.15.`

This approach, using `f'Format {data}'`, is actually a shorthand. The values in the `{data}` brackets come from variables that are available to the program. It's a convenient shorthand for the longer form `'Format {}'.format(data)`. There are a few things to keep in mind about the longer form, but it does have a lot of benefits. Chiefly, by using the `''.format()` form, you can store the template string itself in a variable and pass it around your program. In most large software, this is how we handle translating user interfaces between languages - define all the user readable strings, and then choose the right one while our translators handle the hard work of getting the various forms right!

The `format()` form also differs in that the arguments are positional - you can put as many variables into the `format()` method call as you like, and when the template string is filled in, values are used from left to right in the `{}` blocks. When you use the `f''` format, you always specify the variables that go into each `{data}` block.

## Formatted HiLo

Let's look back at HiLo, but this time using format strings

```
def calc_winnings(number, guesses):
    new_winnings = (6 - guesses) * 10
    print(f"You won {new_winnings} dollars!")
    winnings = winnings + new_winnings
    print(f"Your latest winnings are {winnings} dollars!")
    return winnings

def play_again():
    again = input("Play again? (Y/n) ")
    return again != "n"

def finished(winnings):
    print(f"Thank you for playing! Your total winnings were {winnings} dollars!")
```

That's all there is to it! Pretty easy, right? And I thin it's much more legible and it is definitely easier to work with - no need to `str(num)` all the time!

## Formatting Options

There are a huge number of formatting options we can do. We can set a certain amount of padding, so if we were making a table, the output would always take 10 spaces whether the string was 2 or 6 or however many characters wide. We can format numbers to always have a certain precision after the decimal point, or have certain types of separators between every 3 digits. Rather than listing them all here, I'll recommend visiting https://pyformat.info/ and reading their excellent documentation. (As you can tell, we're using the "New" format.) That said, we will be using some of these as we go further in the book.

## Exercises

- Revisit some or all of your programs and make them use format strings instead of string concatenation.

When you've gotten all your programs using format strings for beautiful output, we can learn about [multiple pieces of data with arrays](../03_arrays/README.md)

# Arrays in Python

In this section, we're going to do some math on sets of numbers. We're going to write a couple functions to determine some statistics about these sets of numbers. Just like chapter 1, the math is NOT the interesting bit here! Focus on the code and the programming, and if you're interested, you can follow up on the math on Wikipedia.

Let's start a new program, `arrays.py`

```
data = range(2, 20, 3)
for i in data:
    print(i)
```

When we run this, we get 6 lines:

```
2
5
8
11
14
17
```

```
`range` is a built in function that gives us an array of integers. The first
integer in the array is `2`, the first number in the arguments. The array will
grow up to (but not including) `20`, the second number. The last argument, `3`,
tells the range how many to increment by each time. This argument is optional,
and defaults to `1`. So if you just want a normal list of 20 numbers, you can
do `range(0, 20)` to get the numbers 0 through 19, or `range(1, 21)` to get
1 through 20. Because we set it to 3, we jump several at a time.
```

```
This is the data we'll use to test the functions, and then we'll get a bigger
data set to answer a real science question!
```

Add this code at the end of the file:

```
[source,python]
```

```
 def sum(my_array):
    total = 0
    for val in my_array:
        total += val
    return total

data_sum = sum(data)
print(f"Data sums to {data_sum:.2f}")
```

```
When we run it, we should see the output from above, and this line as well:
```

```
 ----
Data sums to 57.00
 ----
```

Here, we define a new function, `sum`. It takes one argument, `my_array`, which will be the array of data to operate on. Sum, or summation, is just a really fancy way of saying "Add up all the numbers". We do that by starting the variable `total` with the value 0, and then we use a `for ⋯ in ⋯` loop to get every item. With a for loop, we name the variable in the first area, and then provide the value to loop over as the second (before the `:`). So this `for ⋯ in ⋯` the array to loop over is `my_array` and the variable for each time is `val`.

The body of the loop is one line - `total += val`. We saw `+=` in the rugs and it's just a short way to say `total = total + val`.

The body of the loop will execute one time for every value in `my_array`. Each time before the body executes, python will assign the next value in `my_array` to the variable `val`.

After the loop body (because it's back to the first level of indendation), we return the total. And that's it for the function!

Back at the file level, we run the function using the data we got above, and the print it using a format string. This string uses `{data_sum:.2f}`, which takes `data_sum`, treats it as a `f`loat, and provides 2 digits of precision.

Let's keep going with a function to calculate the average. The average, or mean, of a set of numbers is found by adding all of them up and dividing by the total number of items in the set.

```
def avg(my_array):
    total = sum(my_array)
    length = len(my_array)
    return total / length

data_avg = avg(data)
print(f"Data average is {data_avg:.2f}")
```

The new line of output should be

```
Data average is 9.50
```

This function starts the same way as sum, by giving it the name `avg` and taking one parameter, my_array. But then, instead of adding up all the numbers itself, it just uses the function we wrote in the last exercise. Then it uses the built in function `len` to get the length of the array. Keep `len` in mind, as it's the canonical way in python to get the size of any complex object - arrays or otherwise! At the end, we print out the same way.

We have one last function, which is a lot more complex. The standard deviation of a set of numbers describes their statistical distribution. If the standard deviation is a larger number, it means there is more variability in the sample. The numbers are more different to each other than similar. If the standard deviation is smaller, the numbers are more clustered together.

> The formal definition of the standard deviation involves taking all the numbers in the array, squaring the difference in the average of all values from the single value, summing all those squares, dividing that sum by the number of values in the data set, and finally taking a square root of the whole thing. Ask your nearest stats professor for details, and trust the math part of the code below :)

```
import math
def std_dev(my_array):
    my_array_average = avg(my_array)

    diffs_squared = []
    for val in my_array:
        diff = val - my_array_average
        diff_squared = diff ** 2
        diffs_squared.append(diff_squared)

    sum_of_squares = sum(diffs_squared)
    length = len(my_array)
    return math.sqrt(sum_of_squares / length)

data_std_dev = std_dev(data)
print(f"Data std dev is {data_std_dev:.2f}")
```

When we run this, we find quite a wide standard deviation:

```
Data std dev is 5.12
```

Which makes sense. We don't have a lot of numbers, and they have a bit of space between them. Let's get a real data set, and try to answer an [old wive's tale](https://www.bbc.com/future/article/

20171127-the-truth-about-three-childbirth-myths): "First babies arrive late". We'll be using a dataset taken from the Unite States Center for Disease Control's 2002 National Family Growth Survey (or US CDC 2002 NFGS) for short. This survey captures a myriad of details about tens of thousands of pregnancies in the United States in the one year alone. From that data set, I've create a small file you can download and save next to your `arrays.py` program: `data.py`

Once this file is saved, go back to `arrays.py` and add

```
from data import PREGNANCIES

# Set up the lists to calculate sums on
preg_weeks_first_babies = []
preg_weeks_second_babies = []

for pregnacy in PREGNANCIES:
    order = pregnacy[0]
    weeks = pregnacy[1]
    if order == 2:
        preg_weeks_second_babies.append(weeks)
    elif order == 1:
        preg_weeks_first_babies.append(weeks)

print(f"First babies are born at an average of {avg_first_babies:.2f} weeks.")
print(f"First babies have a standard deviation of {std_dev_first_babies:.2f} weeks")
print(f"Second babies are born at an average of {avg_second_babies:.2f} weeks.")
print(f"Second babies have a standard deviation of {std_dev_second_babies:.2f} weeks")
```

Let's find out if the old wive's tale holds up to statistical scrutiny

```
First babies are born at an average of 38.58 weeks.
First babies have a standard deviation of 2.68 weeks
Second babies are born at an average of 38.64 weeks.
Second babies have a standard deviation of 2.72 weeks
```

The verdict is in - first and second babies have some difference on the order of hours, certainly not the weeks claimed in the anecdote we're questioning.

Let's take a closer look at this code we have above. First, we get the data that we are interested in by using `from data import PREGNANCIES`. We've see simple import statements before, like `import math`. This let us use `math.sqrt`, but also gave us access to `math.sin` and `math.atan2`. If we don't ever use the trigonometry functions, why would we want to import all of them? The same thing with this file- rather than importing the whole thing and asking for `data.PREGNANCIES`, it's way easier to just import the single thing we want. So we use this `from ⋯ import ⋯` form and fill it in with the file or library, and the individual value that we want from that file.

Now that we have our pregnancies, we're going to keep track of two arrays, one for first babies and another for second babies. We saw with `range()` that we could create an array with integers already filled in, but now we're going to start with an empty array (just like we started with empty string).

The empty array is `[]`, which says "This is a big long list that we can add things to later". Later we will add to it not by using `+` like with trings, but instead by calling `append` which is a function the array has to add data to it.

With our data keeping set up, we start our for loop over the pregnancies array. If you looked at the data file, you may have seen that it's actually a bunch of arrays inside a big one. In python, we call a small fixed-length array (like these ones, with two elements) a tuple. This is useful for exactly this case - we want to keep pairs of data about each pregnancy in our list. The pieces of data here are first, the order of the pregnancy (so whether this is the woman's first or second or fifth pregnancy), and second, the number of weeks into the pregnancy she gave birth.

Inside the loop, we create two variables, `order` and `weeks`. Because each item in `PREGNANCIES` is an array, we access the first and second elements respectively. To do so, we use the `[index]` array access notation. We write the variable which has the array, `pregnancy`. We then add `[]` immediately after it. Finally, we use a variable or value to access the item we want - remember, indexes start at zero. So `pregnancy[0]` gives us the first item in the tuple (the array of two items), which our data file defines as the order of the pregnancy. `pregnancy[1]` gives us the weeks' duration of of the pregnancy, again which we know from the data file.

The loop body is several `if ⋯ elif ⋯` block. It looks at the order, and decides which of the lists we're tracking it should append with the length of the next pregnancy. If the order is great than 2, we will ignore it because we are only interested in either a woman's first or second pregnancy. Then if the order is two, we append the length to the array of numbers of weeks in a second pregnancy. If the order is one, we put it in the other array.

After we've split the `PREGNANCIES` data set up this way, we exit the `for` loop and pass our two arrays into our functions we had defined for `avg` and `std_dev`. With these results back, we print them out for the user!

## Exercises

1. **Statistics**
    a. Using the `sum` function above, trace the execution of these programs:
        i. `print(sum([15, 15, 15, 14, 16]))`
        ii. `print(sum([2, 7, 14, 22, 30]))`
    b. Using the `avg` function above, trace the execution of these programs:
        i. `print(avg([15, 15, 15, 14, 16]))`
        ii. `print(avg([2, 7, 14, 22, 30]))`
    c. Using the `std_dev` function above, trace the execution of these programs:
        i. `print[15, 15, 15, 14, 16]`
        ii. `print(std_dev([2, 7, 14, 22, 30]))`
    d. In statistics, population **variance** is a part of the standard deviation. Specifically, it's the part of calculating a standard deviation that sums the squares of differences and divides by N, before taking the square root. In other words, the standard deviation is the square root of the variance of the array of data. Rewrite your `std_dev` function to perform the first part of

the calculation, the `variance`, in its own function, and then use that function to calculate the standard deviation.

2. **Recipes** Use arrays to store the ingredients in each recipe. For each ingredient, the array should have a tuple of three items, including the name, the default amount, and the measurement size. Instead of printing each ingredient specifically, use a function which loops over the array to print each ingredient. Don't forget the scale parameters!

When you've completed the exercises, we can wrap up this chapter in the textbook.

# Functions, Arrays, and Strings

## Objects and properties

In most of our Python up to this point, we've been using single variables to refer to single values. We have one variable that refers to an `area`, which we know to keep in mind as an area of a rug based on context. We have one variable `n` which we set to `len(my_array)` to track the size of an array. We have one single variable that we set a couple times to hold the perimeter cost of a rug.

If we didn't have scopes, this would be ridiculous - every different function that wanted to talk about an `area` would need to come up with a unique name for its variable! Whether its rugs or the size of a floor in your house, you'd end up with something like `area_rug_when_computing_cost` and `area_rug_from_user_input`! Scopes at least let us have one `area` within the function `rug_cost`, and we can know by context what it's being used for.

But keeping all these variables on their own is kind of a pain - a rug doesn't just have an `area`, it also has a `perimeter` and a `side_length`. In the programs in this chapter, we're going to learn about another way to group bits of relevant data - **objects**.

Objects are bundles of values that are all related to one another in some contextual way. For the rug, an object would keep track of all the values for its side length, whether it has fringe, its area and perimeter and cost, and whatever else we need to access from it. Once we put a rug object together, we can have one variable `rug` and get all those values from it.

We structure the object using **properties**. Properties are variables that are tied to a single object. Just like a variable, they have an **identifier** and a **value**. Unlike variables, they aren't bound to a scope, but rather are stored with an object. Just like variables, their values can be whatever is needed — a number, a string, an array, or even another object!

Before we go back to our rugs example, we're going to work with a built-in object type first, `datetime`.

### Dates

In computer programming, handling dates can get complicated. Unlike numbers, dates have dozens of rules controlling how a date is handled. From different numbers of days in months, to leap years, to "11th" compared to "21st", dates require much special handling from computer programmers.

In Python, the basic date handling is in a module called datetime. We're going to start by actually grabbing two items from it, `datetime` and `timedelta` which we'll explain in-depth later. (While we're here, we'll also grab our trusty [`colors.py`](./colors.py) - download it again or copy it if you are in a new folder).

Make a new file, `calendar.py`, and let's get going!

```
import colors
from datetime import datetime, timedelta
from time import sleep
```

These lines allow us to use `datetime` later in the program. It's similar to `math` from chapter 1. For us, we want the function `datetime.now()`, which gives us a "date" thing, seperate from an integer, float, or string, that allows us to check the month, day, and year.

A `datetime` is an object with properties for each of the things you might expect - `hour` gives us the numerical hour in the day that that datetime represents; `month` the month of the year, `microsecond` the number of millionths of a second the datetime was created at.

When we have a single `datetime`, these values don't change unless we tell it to! When we used the function `datetime.now()`, it looked at the computer's clock **at that moment in time**, made a note of it, and that moment **at a point in time** is what we can look at. If we want a different `datetime`, we have to call `datetime.now()` **another time**, and this time we get back a **different** datetime! It doesn't update the one that we had gotten before. Let's see what that looks like.

```
moment_1 = datetime.now()
sleep(0.5)
moment_2 = datetime.now()
sleep(1.5)
moment_3 = datetime.now()

print(moment_1.hour, moment_1.minute, moment_1.second, moment_1.microsecond)
print(moment_2.hour, moment_2.minute, moment_2.second, moment_2.microsecond)
print(moment_3.hour, moment_3.minute, moment_3.second, moment_3.microsecond)
```

Running that at a little before 3 pm I got this output:

```
14 48 36 731726
14 48 37 235577
14 48 38 736215
```

You should have something similar - the second one is about .5 seconds after the first, and the third is about 2 second after the first!

> `datetime` tracks hours using 24 hours in a day. Later in this section, we'll see how to get AM/PM hours.

#### Desktop Calendar

Before we dive in to the rest of this program, let's take a look at what it'll show us. It's going to be a program that prints a calendar of the current month. The days of the week will be along the top, with weekends a different color. It will also mark today a specific color, as well as any holidays we have this month. At the bottom, it will print the number of days as well as the number of workdays

until the next holiday, and finally it'll have a digital clock printing the time in H:M format.

It will look like this when we're done:

![Desktop Calendar](./desktop_calendar.png)

Let's take a look at a function that returns the string for which month of the year a date is. Remove everything but the imports above, and add this instead. The `def` keyword starts starts **def**ining a section of code. The parentheses tell python it is a function, while `month` gives it a name. Finally, `date` names an argument for the function. This will take a `datetime` object, not just a number!

```
def month(date):
    if date.month == 1:
        return "January"
    elif date.month == 2:
        return "February"
    # 10 more cases for you to write out

print(month(datetime.now()))
```

Running that this morning, it told me it was `May` as expected!

In this function, we have the argument date. We expect callers to pass a datetime as the argument. One property of a datetime is `month`, which gives us the numeric month of the year starting from 1 as January. We access a property on an object using a `.`. So `date.month` says go to the value in the variable `date`, find its property `month`, and use **that** value for our expression.

At this point, we can do the top bar of our calendar. Replace the print line with this bit of code.

```
moment = datetime.now()
month_face = f"{moment.day} {month(moment)} {moment.year}"
print(f"{month_face:^28}")
```

And running that, we see

```
        19 May 2020
```

Let's look at this bit by bit. We get a single moment, so that we can use that one moment in time several times, once to get the day, once for the month, and once for the year. We get the `day` as a property of `moment` using `.day`, and the same for `.year`. We use the `month` function we just wrote to get the full name of the month, rather than the number.

We start a new variable, `month_face`. This is so we can easily center it, on the next line. We build it by using a **template string** which we introduced in the strings section last chapter. We know it's a format string because it has an `f` right in front of it. It accesses two different properties on the object returned by the `no` function (which, again, is a datetime) — `day` and `year`. It also takes the entire object and passes it to our `month` function we just wrote above.

We store this in the variable `month_face` so that we can use a second format string to center the text we just created. We do this with the `:\^` format specifier. In chapter 2, we used `:.2f` to specify the amount of precision we wanted in a floating point number. In the format string, the `:` marks where the variable ends, and when modifiers to the formatting begin. There are three that are useful when working with text, and those are `:<`, `:\^`, and `:>`. All three take a number after the arrow, and they fill out the text with extra spaces on the sides so that the final string has many characters (including spaces) as the number specifies. The text is padded with whitespace after, around, or before the value it got from the left of the `:`. That is to say, these all let us left, center, and right justify a string of text. In this case, we use 28 characters because that's how wide our calendar will be.

#### Calendar Days

Let's start to build out the calendar days part of the program. We're going to do it in stages, starting simple and adding to it as we go. We still want to get a plan of action before we start.

Thinking through what the calendar should look like, it should have 7 columns, one for each day of the week. It will have 4 to 6 rows, depending on how many weeks the month takes up. Each day in the month, then, will end up having a column, based on which day of the week it is. It will also have a row — but the row its in will depend on how many weeks have come before it at that point. In a python date, the `weekday()` function will give us a number between 0 and 6 — 0 means Monday, 1 means Tuesday, up to 6 being Sunday. So if this gives us the column to put the day in, what about the row?

For the row, we can just keep track of what row we're currently in, and increment it by one each time we get to a Sunday. With this plan, we need to do a couple things. We need to loop through all the days in a single month, tracking which week of the month ourselves and increasing it every time we get to a sunday. Then for each day in the month, we need to print it out using the weekday and week of the month to choose a row and column to print it in. Two separate things to do means two separate functions! Let's write ourselves a `print_day` function first, which takes a `datetime` and week to print it in.

```
def print_day(date, row):
    column = 2 + 4 * date.weekday()
    print(colors.at(row, column), end="")
    print(date.day)

print_day(datetime.now(), 1)
```

For me, on Tuesday the 19th, I see a few spaces and then `19`.

```
     19
```

`date.weekday()` is a function which will return the weekday (0 is Monday, 6 is Sunday) for the date it's attached to. For our calendar, we want each weekday to take up 4 spaces, so that's why we multiply by 4 - to skip past the other earlier days that week. The `2 +` is just an initial padding to not squish everything up against the left side of the screen. With that row and column, we print

`colors.at()` to move the cursor, but we also add this new piece `end=""`. Usually when we print, it moves the cursor down to the next line immediately. We don't want that to happen, and suppress that behavior using `end=""`.

> The reason it's called `end` is because normally, python uses the special string `"\n"` to mean "New line", and inserts it automatically at the end of every print string. You can think of it as every time you print, you are really doing `print(my_string, end="\n")`. Because 9 times out of 10 this is the behavior we want, it's much easier to do it by default instead of requiring every time we use print to include it. This is the escape hatch for when we don't want the default.

#### Calendar Weeks

Now that we can print one day, let's do the `print_month` function which will print an entire month for us. Delete the `print_day` testing line, and then add & test this `print_month` function.

```
one_day = timedelta(1)
def print_month(date, first_week_row):
    day_in_month = datetime(date.year, date.month, 1)
    week = 0
    while day_in_month.month == date.month:
        print_day(day_in_month, first_week_row + week)
        if day_in_month.weekday() == 6:
            week += 1
        day_in_month += one_day

print(colors.CLEAR)
print_month(datetime.now(), 2)
```

There's a bit more to this one, as we might expect when we need a loop. The function declaration looks about the same as for `print_day` - a `date` to tell us which month we're interested in, and `first_week_row` which we will use to know how far down the page to print the month part of the calendar. The loop will track `day_in_month`. Each iteration of the loop, we add one day to it, and keep looping until the tracking variable is no longer in the same month as the original `date`. Each time through, we also track the current week within the month, incrementing it after printing each Sunday.

Above that, though, we have a constant - `one_day = timedelta(1)`. What do you think that is?

Where a `datetime` tells us a specific point in time, `timedelta` describes the amount of time between two points in time. There are two ways to make a `timedelta`. The first is to subtract one date from another. (No need to type this next example yourself.)

```
time_then = datetime.now()
sleep(10)
time_now = datetime.now()
time_difference =time_then - time_now
print(time_difference)
```

Prints out

```
0:00:10.001196
```

Which tells us it took 10 seconds and some change between those two datetimes we created!

Once we have a timedelta, we can go the other way around.

```
print(time_now)
print(time_now + time_difference)
```

```
2020-05-20 08:08:59.351762
2020-05-20 08:09:09.352958
```

That works great for things like "How long until Christmas Day?" by creating a datetime for the holiday, and subtracting today. We're going to use it a bit differently, by creating a datetime and timedelta directly from the calendar day we're trying to print out.

Quick recap of that: we have the date that we want to use to print a calendar for the surrounding month. The caller of the printMonth function provides that. In the printMonth function, we want to make a new datetime that corresponds to the first day of the month. Then, we want to print that day (not the one the user provided), and loop through all the days in the month. We get the next date in the month by creating a timedelta which represents the concept of one day, and then add that to the date representing the start of the month.

That bit alone looks like this:

```
one_day = timedelta(1)
def print_month(date):
    monthly = datetime(date.year, date.month, 1)
    while monthly.month == date.month:
        print(monthly.day)
        monthly += one_day
```

Running this with `print_month(datetime.now())`, I get this output:

```
1
2
3
4
... 25 more lines...
30
31
```

So we have `one_day` which is the `timedelta(1)` to represent a single day. That is defined at the file level - we'll just use the same one every time we call the function. We need a new datetime that represents the beginning of the month, because the provided `date` variable is going to be any arbitrary point of time within a month. We get that instance of the beginning of the month by creating a new `datetime` with the same year and month as the one we're given, but then we set the `day` to `1` (the 1st), and don't provide any values for the hours, minutes, or seconds. This is the `datetime` which represents the point in time at midnight of the first day of the month that our input date is in.

We're going to take this datetime object, and keep adding one day to it. As long as it's still inside the month we started with, we'll print it out. The `datetime` will handle the logic itself for knowing how many days to use. We just need to look at the `month` property every timearound.

Looking back at the first `print_month` function up above, we should now be able to make sense of `week` and `first_row`. `first_row` is an argument used to tell `print_month` how far down the page to start printing. `weeks` then tracks which week of the month the loop is in, incrementing it whenever `monthly.weekday()` is `6`, Sunday (incrementing **after** printing).

Run it again, and take some time to think through what we've just covered and how it works together.

#### Highlighting Days

Now that we have our calendar printing out, let's make it highlight each day based on whether it's a weekday, a weekend, a holiday, or the current day. We will do this by first writing a helper function for each of those cases, and then by using those functions in the `print_date` function we already wrote.

```
def is_weekend(date):
    # Monday is 0, Sunday is 6
    return date.weekday() > 4

HOLIDAYS = [
    datetime(2020, 1, 1), # New Year's Day
    datetime(2020, 1, 20), # MLK Jr
    datetime(2020, 2, 17), # George Washington's Birthday
    datetime(2020, 5, 25), # Memorial Day
    datetime(2020, 6, 3), # 4th of July Holiday
    datetime(2020, 9, 7), # Labor Day
    datetime(2020, 10, 12), # Columbus Day
    datetime(2020, 11, 11), # Veteran's Day
    datetime(2020, 11, 26), # Thanksgiving Day
    datetime(2020, 12, 25), # Christmas Day
]

def is_holiday(date):
    return datetime(date.year, date.month, date.day) in HOLIDAYS

def is_workday(date):
    return not is_weekend(date) and not is_holiday(date)
```

The first helper, `is_weekend`, is very easy. It just checks if the `weekday()` is greater than 4 - remember, 0 is Monday and 5 is Saturday!

`is_holidays` does a couple tricks that are new to us. First, it uses a block of data that explicitly lists out the days that are considered holidays. While we could write functions to calculate most of these (Thanksgiving is on the third Thursday of November; Christmas is the weekday closest to the 25th of December) I think it's easier to just list them out. Besides, these holidays are only correct for the US - by making it a block of data, we can just come along later and change it to whatever days we need to for our locale. This type of data driving programming is more common than you might expect!

The `is_holidays` function is then going to use the `in` operator to determine whether a certain value appears in the array at all. `5 in range(1, 10)` would return true, as would `"def" in "abcdefghi"`. But remember that a date is a single point in time, so we normalize whatever date it is to be midnite exactly of the date in question. This type of normalization is another very common trick!

The last function, `is_workday`, almost reads like plain English. Workdays are any day that aren't a weekend or a holiday. Maye we should add vaction days in here as an exercise...

With these helpers in hand, we can expand our `print_day` function with some color base on which of these conditions are true.

```
def print_day(date, row):
    now = datetime.now()
    column = 2 + (4 * date.weekday())
    print(colors.at(row, column), end="")
    if date.day == now.day:
        print(colors.YELLOW, end="")
    elif is_holiday(date):
        print(colors.PURPLE, end="")
    elif is_weekend(date):
        print(colors.RED, end="")
    else:
        print(colors.WHITE, end="")
    print(date.day)
```

Using this in place of our `print_day` before, and testing using the `print_month` function, should get us pretty close to the output we demoed at the beginning of the chapter! We just need to add a couple pieces of "pretty printing" before calling `print_month`.

```
def print_calendar():
    now = datetime.now()
    print(colors.CLEAR, colors.at(1, 1))
    print(" M   Tu  W   Th  F   Sa  Su")
    print_month(now, 3)

print_calendar()
```

All we do here is grab the current datetime, clear the screen, and start writing the calendar at the 4th row, first column. We add a row of the day of the week, and then use `print_month` starting at row 6 to give us the days in the calendar appropriately aligned. It should look like this, all said and done:

```
 M   Tu  W   Th  F   Sa  Su
                 1   2   3
 4   5   6   7   8   9   10
 11  12  13  14  15  16  17
 18  19  20  21  22  23  24
 25  26  27  28  29  30  31
```

## Recap

In this section we looked at using two builtin Python objects, `datetime` and `timedelta`. An object is a programming tool to group related data together. Instead of having a bunch of variables that are near eachother and maybe have the same name, we instead create an **object** that keeps all those values in its **properties**. We can then use the single value of the object, rather than needing to specify each piece of it separately every time.

Up next, we will look at how to create our own objects, when we revisit the rugshop with classes.

### Exercises

1. **Vacations** Add vacation days and give them a color.
2. **Clock face** Look at more properties on datetime, and use them to print a clock face. Bonus: Use the `format` function and Python's datetime format codes.
3. **Days until** Using timedeltas, count the number of days between two dates. Use this and your weekend/vacation checks to print "Days until/Workdays until"
4. **Tick Tick Tick** Using loops and `time.sleep`, make your clock tick by updating every 10th of a second.

# Python Classes

In the last chapter, we wrote pairs of functions to calculate costs for rugs, and to get user input. The function declarations started to get a bit unwieldy and redundant.

```
def square_rug_cost(size, fringe):
def compute_square_rug():
def rectangular_rug_cost(width, height, fringe):
def compute_rectangular_rug():
def circular_rug_cost(radius, fringe):
def compute_circular_rug():
```

The idea of classes will separating each of these concepts into distinct pieces. We can use those pieces in similar and interchangeable ways, and really free ourselves from needing to track a large number of smallpieces by instead tracking a small number of big pieces. Those big pieces can just hang out, do their thing, and let us focus on other areas of our program as it grows.

On the flip side, with object oriented programming and using classes, we do want to put a bit more effor in up front, thinking through what our design will be, before we get down to brass tacks. This up-front thinking will let us potentially shave a lot of time and effort down the road, if it turns out we later built the wrong thing!

Before we start our class, then, let's remind ourselves of what we're trying to achieve. Looking back at our last chapter's rug functions, we have this

```
def square_rug_cost(size, fringe):
    area = size ** 2
    cost = area * 5
    if (fringe):
        perimeter = size * 4
        cost = cost + perimeter * 1.5
    return cost
```

A square rug needs to track two pieces of data - whether it has fringe, and the size of one edge. It then calculates the area, perimiter, and costs from those pieces. Rectangular and circular look similar. Looking at this, we probably want one class, `SquareRug`, which will take two properties, `size`

and `has_fringe`. It will have one method, `cost()`, which uses this same formula but gets the data from its properties. Let's see what this looks like in Python. Type and run the following code in a new file.

```
class SquareRug():
    def __init__(self, size, has_fringe):
        self.size = size
        self.has_fringe = has_fringe

    def cost(self):
        area = self.size ** 2
        cost = area * 5
        if (self.has_fringe):
            perimeter = self.size * 4
            cost = cost + perimeter * 1.5
        return cost

square_rug_1 = SquareRug(5, False)
square_rug_2 = SquareRug(2.5, True)
square_rug_1_cost = square_rug_1.cost()
square_rug_2_cost = square_rug_2.cost()
print(f"Rug 1 costs ${square_rug_1_cost:.2f}")
print(f"Rug 2 costs ${square_rug_2_cost:.2f}")
```

Which gives the same output as the similar version of the functions program.

```
Rug 1 costs $125.00
Rug 2 costs $46.25
```

This is our first class! I think it's time to make a commit!

Let's go through this code and look at the new things we haven't used yet. First we have `class SquareRug():`. Just like a variable or a function, a class has an **identifier**, this one is `SquareRug`. Like variables and functions, the identifiers within the scope they are declared in all have to be unique. Python knows we're making a class, because we say so with `class`. Also like a function the class has a body, which just like a function is demarked with `:` at the end of the line and everything below it indented one more level. We'll cover the pair of parens in the next section.

So we've told Python we want a class named `SquareRug`. We now need to tell it what a square rug is all about. We do that in the body by defining two functions. Because these functions are in a class body, we call them **methods**. The first method is a special method, as you might tell by reading its name. `init` is a special method name that Python will use as the constructor for the class. A **constructor** is a special method that will be called every time we build a new instance of a class, and it is responsible for ensuring all the properties of the class are set up properly.

This `init` function is declared using `def`, just like all our other functions, except at one level of indentation because it's part of a class. The function then takes three arguments, `self, size, has_fringe`. `size` and `has_fringe` make sense, because these are what we expect the class needs for

its data, but what is `self`?

In Python, every method has a necessary first argument which will refer to the instance of the class that the method is invoked on. What that means is that when you have an object, and you go to call one of its methods, python will fill in the first argumemt with the value of the object. This must be marked down explicitly inside the method, and while it's just a variable, by convention we always name it `self`.

So in the constructor, named `init`, we have take three arguments. `self` refers to the object we are setting up. `size` and `has_fringe` come from whatever code asked for a new `SquareRug`. The body of the `init` constructor takes the arguments, and just stores them as properties on `self`.

Just like we can read properties in the calendar program of datetime, so too can we write to properties. We just use the variable name of the object, put a `.`, put the identifier of the property, and use = to assign it like any other variable we've used thus far.

The second method we have, `cost`, should look pretty familiar. The only change is that instead of taking `size` and `fringe` directly, it instead takes `self` and looks up `size` and `has_fringe` as properties on the instance.

Outside the class we have our little bit of test code. We create two square rugs by using the class name as if it were a function, and passing all the arguments **except** `self`, the first. Python handles filling in the first `self` argument on the calling side; it only needs to be explicit inside the method. We create two rugs passing `5, False` and `2.5, True` for the parameters. We call the `cost` methods on each instance, and print out the final value.

We can do this again for the rectangle rug.

```
class RectangularRug():
    def __init__(self, width, length, has_fringe):
        self.width = width
        self.length = length
        self.has_fringe = has_fringe

    def cost(self):
        area = self.width * self.length
        cost = area * 5
        if (self.has_fringe):
            perimeter = self.width * 2 + self.length * 2
            cost = cost + perimeter * 1.5
        return cost

rectangle_rug_1 = RectangularRug(4, 6, False)
rectangle_rug_2 = RectangularRug(2.5, 5, True)
rectangle_rug_1_cost = rectangle_rug_1.cost()
rectangle_rug_2_cost = rectangle_rug_2.cost()
print(f"Rug 1 costs ${rectangle_rug_1_cost:.2f}")
print(f"Rug 2 costs ${rectangle_rug_2_cost:.2f}")
```

With these test cases, we get

```
Rug 1 costs $120.00
Rug 2 costs $85.00
```

This should look and feel pretty similar. We define the new class, `RectangularRug`. Its constructor takes four arguments, instead of three. The cost calculation is a little different. But otherwise, it's all still there!

Take a few minute to write the last rug class, `CircularRug`. Using `5, False` and `2.5, True` you should again get

```
Rug 1 costs $392.70
Rug 2 costs $121.74
```

Let's remove all the testing code and take a look at how we might ask for user input for each of these rugs. Naively, it'll be just like what we had in the functions exercises - ask the user for which type of rug, then ask them all the numbers for that type, and then finally create that rug and print its cost.

```
def price_rug():
    print("1) Square Rug")
    print("2) Rectangular Rug")
    print("3) Circular Rug")
    rug_type = input("Which type of rug? ")
    if rug_type == "1":
        size = float(input("Size of rug: "))
        wants_fringe = input("Has fringe (y/n): "))
        rug = SquareRug(size, wants_fringe == 'y')
    # ... three other types

    rug_cost = rug.cost()
    print(f"The rug costs ${rug_cost:.2f}")

while input("Price another rug? (y/n) ").lower() == 'y':
    price_rug()
```

We now have a control loop and a few classes, so this might be a good time to make a commit. (If you've been making them more often, that's great!)

Right away we see that maybe this isn't the best approach. The whole point of classes was to contain the information necessary to build a rug, but we have this whole function which just asks all the information over again anyway! Let's fix this by moving the code to get the various rug information to inside the classes themselves. We'll still be able to creat a rug of a certain shape and size, but now we'll have a second option to tell the rug to ask for input on its own, simplifying the main loop of our program.

In the SquareRug class, we need to change the constructor to have default values for its arguments, and we need to add a get_values method which asks for input and stores it in the class' properties.

```
class SquareRug():
    def __init__(self, size = 0, has_fringe = False):
        self.has_fringe = has_fringe
        self.size = size

    # The old cost function, it doesn't change

    def get_values(self):
        wants_fringe = input("Should this rug have fringe (y/N)? ")
        if wants_fringe.lower().startswith('y'):
            self.has_fringe = True
        self.size = float(input("Side length of this square rug? "))
```

And do the same thing for the other two classes. Now we can rewrite the control loop, delegating to this new get_values function!

```
def get_rug():
    print("1) Square Rug")
    print("2) Rectangular Rug")
    print("3) Circular Rug")
    rug_type = input("Which type of rug? ")
    if rug_type == "1":
        return SquareRug()
    elif rug_type == "2":
        return RectangularRug()
    elif rug_type == "3":
        return CircularRug()

def print_rug(rug)
    price = rug.cost()
    if rug.has_fringe:
        with_fringe = "with"
    else:
        with_fringe = "without"
    print(f"This rug costs ${price:.2f} {with_fringe} fringe.")

while input("Price another rug? (y/n) ").lower() == 'y':
    rug = get_rug()
    # Ask for inputs
    rug.get_values()

    print_rug(rug)
```

Now the `get_rug` function only needs to know about the types of rugs we have in the program, and chooses one of them. It returns a new instance of the chosen rug type. Then back in the loop, it has

the rug itself ask the user for the values, which fills in the rug. The control loop sends that rug to this new `print_rug` function, to get nice formatting. Now, we have a clean, clear, logical separation of concerns for our rug types!

This feels like a great time to make a commit. You might also at this point want to compare this version of the program to the prior commit, and look back over what you've changed!

## Inheritance

You may have noticed some duplication starting to crop up across these classes. Each class has almost the same implementation of the `cost` method, and each one has to duplicate `get_values` asking for whether the rug has fringe.

As you might have guessed, we have tools to handle this type of complexity! When several classes need to share the same functionality, we can introduce a **base class** to handle the shared pieces, and then each of the classes we have becomes a **derived class** which fills in just the pieces it cares about.

This sounds a little more complicated than it is, so let's walk through what we will do to get this separation before we look at it in practice. What we're going to do is create a new class, `Rug`. `Rug` will do all the things our other rugs can do, but it's not going to have any logic that's specific to a rug shape. It will have a constructor, which will only take whether it has a frince (which applies to all rugs). It will have a method that calculates cost, and one that asks for `get_values` that asks whether the user wants fringe. But instead of doing the calculation for an area and a perimeter, it will instead have a new pair of methods `area` and `perimeter` to extract those calculations out. All together, our base `Rug` class looks like this:

```
class Rug():
    def __init__(self, has_fringe = False):
        self.has_fringe = has_fringe

    def get_values(self):
        wants_fringe = input("Should this rug have fringe (y/N)? ")
        if wants_fringe.lower().startswith('y'):
            self.has_fringe = True

    def area(self):
        return 0

    def perimeter(self):
        return 0

    def cost(self):
        area_cost = self.area() * 5
        if self.has_fringe:
            perimeter_cost = self.perimeter() * 1.5
        else:
            perimeter_cost = 0
        total_cost = area_cost + perimeter_cost
        return total_cost
```

We can't really run this yet, because any time we create a `Rug` its cost will always come out as `0`. Instead, we need to tell Python that `SquareRug` and the rest want to **derive** or **inherit** from this class' definition. What that means is that any class we have derived will have access to these functions and behaviors, but more importantly, it is able to replace the definitions as needed! It also can use all the properties the parent class (the one it derives from) has, as well as being able to declare new properties specific to it.

In practice, this means that our `SquareRug` class will need to add properties for its size, and provide new definitions for the `area` and `perimeter` methods. I'll give the example of `SquareRug`, and you can do the same for `RectangularRug` and `CircularRug`.

```
class SquareRug(Rug):
    # __init__ and stays the same and is unchanged

    # remove the cost method

    # change the get_values method
    def get_values(self):
        self.size = float(input("Side length of this square rug? "))

    # add these methods
    def area(self):
        return self.size ** 2

    def perimeter(self):
        return self.size * 4
```

Here, we still have our class identifier, but we added the `Rug` parent class to the inside of the first parenthesis. This is what tells python this is a derived class. The `get_values` method asks for the side of the rug, and the `area` and `perimeter` methods do just the specific calculations for the square.

If we execute our program now and choose a square rug, the `get_rug` function will create a new `SquareRug` object. Then, when we call `cost` in `print_rug`, it will look on the `SquareRug` for a `cost` method. Because there is no `cost` method defined on `SqaureRug`, it will look on the super class, `Rug`. `Rug` does have a `cost` method, so Python will execute that method using the instance of the `SquareRug` as the `self` argument. As `Rug::cost` executes, it will ask for the `area` method on the `SquareRug` instance. `SquareRug` **does** define an `area` method, so Python will use that version which returns the size squared.

Update the Rectangular and Circular rugs to use the base class functionality!

When you're done, make a commit! This should be much cleaner, and would be another great time to review the differences from the last check point.

#### Accessing base behavior

But wait, this is missing something - how will it get whether the user wants fringe? Because we replaced the `get_values` method, this new version will get called which will only ask for the side length of the rug! What we want in this case is to call a **super** method. That's a fancy way of saying we want to get the original version of the `get_values` method, before we changed it, and use that instead.

```
class SquareRug(Rug):
    def __init__(self, size = 0, has_fringe = False):
        Rug.__init__(self, has_fringe)
        self.size = size

    def get_values(self):
        Rug.get_values(self)
        self.size = float(input("Side length of this square rug? "))

    # Other methods stay the same
```

These two methods first get access to the method on the parent class by asking for it specifically - `Rug.init` and `Rug.get_values`. This treats the method specifically as a function, just like calling any other function. However, because we're accessing the method directly rather than through an object, we need to provide the `self` argument explicitly. This all results in calling the `Rug's version of `get_values` on our rug instance to ask for and set the `has_fringe` property, beore doing `SquareRug's user input for side length. For completeness and consistency, we do the same thing in the `init` constructor - instead of setting `has_fringe` at this point, we let the super class constructor handle it. While it's a bit silly for this simple case, it's a good habit to get into for when the base class has more complex initialization behavior.

Let's look at this for `RectangleRug`, and then you can do it yourself for `CircleRug`!

```
class RectangularRug(Rug):
    def __init__(self, length = 0, width = 0, has_fringe = False):
        Rug.__init__(has_fringe)
        self.length = length
        self.width = width

    def get_values(self):
        Rug.get_values()
        self.length = float(input("Length of this rectangular rug? "))
        self.width = float(input("Width of this rectangular rug? "))

    def area(self):
        return self.width * 2 + self.length * 2

    def perimeter(self):
        return (self.width + self.length) * 2
```

Now this is looking like it's saving us some code! Our control loop and functions for choosing and printing a rug are still working, because we didn't change how any of the types of properties or methods on the object! We only changed the implementations of the methods themselves. This process, taking an existing program and rewriting parts of it to be more legible or efficient, is called **refactoring**. We will do it many more times!

There's one more refactoring I'd like to do in this program before we move on to the next section. There's no reason the rugs can't print out themselves. They already ask for input, and if we have

them print their own output, we can easily say what type the rug is!

We'll do this in two parts. First, we'll add a method `print` with no arguments to the base `Rug` class. Then we'll add a property `description` to each rug class.

```
class Rug():
    def __init__(self, has_fringe = False, description = ""):
        self.has_fringe = has_fringe
        self.description = description

    # ... The other methods, unchanged

    def print(self):
        price = self.cost()
        if self.has_fringe:
            with_fringe = "with"
        else:
            with_fringe = "without"
        description = self.description
        print(f"This {description} rug costs ${price:.2f} {with_fringe} fringe.")

class SquareRug(Rug):
    def __init__(self, size = 0, has_fringe = False):
        Rug.__init__(self, has_fringe, "square")
        self.size = size

    # Other methods
```

I think you get the point, and can handle `RectangularRug` and `CircularRug` on your own!

Now that `print` is in the parent rug, we can remove our entire `print_rug` function. Then, we need to replace the call in `price_rug` with a call to `rug.print()`:

```
def price_rug():
    rug = get_rug()
    rug.get_values()
    rug.print()
```

Here's a sample session:

```
Price another rug (Y/n): Y
1) Square Rug
2) Rectangular Rug
3) Circular Rug
Which type of rug? 1
Should this rug have fringe (y/N)? N
Side length of this square rug? 5
This square rug costs $125.00 without fringe.
Price another rug (Y/n): Y
1) Square Rug
2) Rectangular Rug
3) Circular Rug
Which type of rug? 3
Should this rug have fringe (y/N)? y
Radius of this circular rug? 5
This circular rug costs $439.82 with fringe.
Price another rug (Y/n): n
Goodbye!
```

When you're done with all three, let's make our final commit and compare it to our first version with the original control loop. Quite a change, hun?

## Exercises

- Change the price per square foot and the price per perimeter.
- Refactor `SquareRug` to extend from `RectangularRug` instead of `Rug`. The user interface (the visible text the user sees and inputs the user gives) should not change.
- Allow the user to specify the color of each rug. Colors do not change the price of a rug.
- Incredi-Rugz has added a new type of rug, the Peackock rug. This rug comes in small, medium, and large sizes; it never has fringe; and it only comes in one color, "Peacock". Small peacock rugs cost $125; medium cost $250, and large cost $500.
- The rugs program currently asks for several rugs, prices each, and then exists. Write a new class, `Order`. Move your control loop into `Order`, which should:
- Ask for the name of who's making the order
- Repeatedly ask whether the user wants to add a rug to the order
- If the users wants to add a rug to the order, use the same prompts to create and describe the rugs.
- When the user is done asking for rugs, print out the order including who is placing the order, a summary of each rug with type, size, color, and price, and a total cost for all the rugs in the order.

In this section, we've looked at how we define and create our own classes. Classes describe objects, bundles of data with certain properties and methods that work hand-in-hand. We rewrote our Rugs programs, and saw how using classes and objects let us finally have much less duplication and much clearer flow of operations & data. Taken together, objects allow us to abstract data in the

same way functions allow us to abstract operations and code execution.

### Exercises

1. **Recipe Book** Create classes for your recipes program in chapter 1. At a minimum, you should have classes for `Recipe`s and `Ingredient`s. Both should have a `print_for_desired_servings` method, which takes the number of people to cook the recipe for, and prints the scaled ingredients for that many servings.

    a. **Measurements** Write a class hierarchy (base class and sub classes) for ingredient measurements. Ingredients should take a name and a measurement. Measurements should take an amount. When you write method to scale the measurement to fit the serving, it should return a suitable string. For instance, convert 1/8 cup to 2 tablespoons; or 1 tablespoon and 1 teaspoon to 4 teaspoons.

Now that we know the basics of programming with objects, we can look at tracing objects in memory.

## Maze Design with Classes

As we discussed in the textbook, we're going to build a program that lets us control a character moving through a maze. We will represent this using a variety of classes which interact. The roooms themselves will be instances of a `Room` class, which will have doors to the north, south, west, and east. (In a later exercise, we'll change the room to have an unlimited number of doors.) A door, similarly, will be an instance of a `Door` class. Unlike the room, it will have an ever fixed number of sides. A door can only connect two specific rooms, never more, never less.

So for properties, a room will need a door in each cardinal direction. However, not every cardinal direction will have a door. We represent this case with the special value `None`. `None` in Python is used as a value wherever you need to indicate a missing object. So it's for a variable or property that we would expect to have a value that's an object type, but instead it isn't present. We then can look for when a value is `None`, and behave accordingly - in this case, by saying there's not a door in the north or west walls.

Let's start writing out this code. I'm going to do this for the base features as well as the north door. You will need to fill in the east, south, and west doors in your own `maze.py` file.

```
class Room():
    def __init__(self, description, key = None):
        self.description = description
        self.key = key
        self.north_door = None

    def add_north_door(self, north_room, lock = None):
        self.north_door = Door(self, north_room, lock)
        north_room.south_door = self.north_door

class Door():
    def __init__(self, side_a, side_b, lock = None):
        self.side_a = side_a
        self.side_b = side_b
        self.lock = lock

    def other_side(self, side):
        if side == self.side_a:
            return self.side_b
        else:
            return self.side_a

entry_room = Room("the entrance")
exit_room = Room("the exit")
entry_room.add_north_door(exit_room)

print(exit_room.south_door.other_side(exit_room).description)
```

When you run this code, it should print out `Entrance`, which is the room on the other side of the south door from the exit room. The last line of code looks a bit onerous right now, but we'll be replacing it with something more sensible in a moment.

Disecting the code we just wrote, we have two classes, one each for `Room` and `Door`. Neither extends another class, so we have nothing in the parens. Both classes have a **constructor** , which in Python is a method `init`. The room takes a `description`, a string that describes the room, and `key`, which is the key (if any) the room has. We use the default value `None` because we expect it's more common for a room not to have a key. The room needs the two sides, and also a lock. It has `None` as a default value for the same reason as the `key` in the `Room`.

Both classes take their constructor arguments and put those values into properties on the objects that the class constructor created.

`Room` has a method to add a door connecting a room to the north. Rather than requiring the program to create the door and then attach it themselves, which would be three steps for the programmer, we'll instead have a single method for each door direction will will handle creating the door, and attaching it to both rooms. This type of **object creation** is common because it reduces the number of things necessary to remember to do to create a correct object.

## Locks and Keys and Keyrings

In the next section of the code, we need to look at how the locks and keys fit together. We're actually going to make two classes here - one to represent the locks themselves, and the second to represent a keyring as a whole. Later in the program, we'll give the player a keyring and use that to check whether some doors can or cannot be traveresed.

```
class Lock():
    def __init__(self, fitting_key = None):
        self.fitting_key = fitting_key

    def accepts(self, key):
        return key == self.fitting_key

class Keyring():
    def __init__(self):
        self.keys = []

    def add_key(self, key):
        if not key in self.keys:
            print(f"Picked up {key} key")
            self.keys.append(key)

    def can_unlock(self, door):
        if door.lock == None:
            return True
        for key in self.keys:
            if door.lock.accepts(key):
                return True
        return False

entry_room = Room("entrance")
exit_room = Room("exit")
blue_lock = Lock("blue")
entry_room.add_north_door(exit_room, blue_lock)
keyring = Keyring()

print(keyring.can_unlock(entry_room.north_door))
keyring.add_key("blue")
print(keyring.can_unlock(entry_room.north_door))
```

A lock keeps track of which key fits it, and provides a simple method to check whether it accepts a certain key. Having this as a separate method allows more flexibility down the line - while today we have locks which take a single key, we could think of ways to extend the `Lock` class to require one of several keys (like a specific key and a master key), or make it need multiple keys to unlock.

The keyring tracks which keys have been found, and for any given door, it asks the lock on the door if any of its keys fit. If they do, the `can_unlock` method returns `True` and we know this `Keyring` lets us through that door.

## Game and Player

The last two pieces before our game works are a `Player` and a `Game`. `Player` tracks where the player currently is in the maze, keeps a `Keyring` to unlock doors, and a method `move_through` to check if the door is open and then update which room the player is in.

```
class Player():
    def __init__(self, start_room):
        self.keyring = Keyring()
        self.current_room = start_room

    def move_through(self, door):
        if door != None and self.keyring.can_unlock(door):
            self.current_room = door.other_side(self.current_room)
            self.current_room.on_enter(self)
        else:
            print("Cannot move through that door.")
```

The player starts itself in some given room, and makes itself an empty keyring. The `move_through` method takes a door that the player should attempt to go through. If the door doesn't exist (is `None`, or a missing object), or if the keyring dosn't unlock the door, nothing happens. If the door is present and can be unlocked by the keyring, the player does move through the door. It updates the current room to be whichever room is on the other side of the door from where they started. The last line is a call to a method we haven't seen yet - `on_enter`.

The idea here is that, when a player enters a room, the room has an opportunity to tell the player object anything that it needs to know. In this case, we're going to let the room tell the player about the key it has. Let's add this method to the room class.

```
class Room():
    # ...

    def on_enter(self, player):
        if self.key != None:
            player.keyring.add_key(self.key)
```

This type of programming is called **inversion of control** which is just a fancy way to say one object passing itself to another object to let the other object decide what to call and which properties to change on the first.

Our final class will be the `Game`, which wraps all the rooms and players up and handles user input and output. Let's type it in, and then we'll review it. Just like the first `Room` class, this code includes the north door. You will need to do the other four directions.

```
class Game():
    def __init__(self, start_room, end_room):
        self.player = Player(start_room)
        self.end_room = end_room

    def print_room(self):
        room = self.player.current_room
        print(f"You are in {room.description}.")
        if room.north_door != None:
            print("There is a door to the (n)orth.")

    def move_player(self, direction):
        room = self.player.current_room
        if direction == "n" and room.north_door != None:
            self.player.move_through(room.north_door)
        else:
            print("No room in that direction.")

    def play(self):
        while True:
            self.print_room()

            direction = input("Which direction do you go? (q to quit) ")[0].lower()
            if direction == "q":
                break

            self.move_player(direction)

            if self.player.current_room == self.end_room:
                print("You won the treasure!")
                break
```

A `Game` needs the starting and ending room of the maze. Instead of storing the `start_room`, it instead creates a new `Player` which starts in the `start_room`. The `Game` does store the `end_room`, to check during the `play` method when the player has won.

The bulk of `Game` is in `play`. Like the HiLo game or the rug calculator, a `while True:` loop keeps repeating the game steps over and over. The steps are intended to be concise, using the helper methods for the working aspects. The `Game` prints out the room the player is currently in. It asks which way the user wants to go. If the user decides to quit, the loop exists. Otherwise, have the player move in the input direction. After they have moved, check if they won - and if so, congratulate them and exit! If they're not at the exit yet, loop back and play some more!

Let's try it out on a bigger maze:

```
room_a = Room("the entry room")
room_b = Room("the library", "blue")
room_c = Room("the dining room")
room_d = Room("the kitchen")

room_a.add_east_door(room_b)
room_a.add_north_door(room_c)
room_c.add_north_door(room_d, Lock("blue"))

game = Game(room_a, room_d)
game.play()
```

This last block builds out a full game! We have four rooms, a locked door, and start the player in the entry. Give it a shot!

## Exercises

1. Make a bigger maze!
2. Add more items or types of keys?
3. Provide descriptions of doors.
4. Make rooms support as many doors as you want to give them.

#### Recipe book

Write a program that shows a recipe book. This recipe book will let users choose a recipe from a list of recipes, and then when the user is reading a recipe, let them change the number of servings. When the user changes the number of servings it will update all the measurements as needed. A recipe book has a title and many recipes. Each recipe has a title, a number of servings in its written instructoins, a list of ingredients, and instructions as a string or list of directions. Ingredients in a recipe will need a name, and also some notion of a measurement.

Measurements will be where a lot of the magic in scaling the number of servings. A measurement could be as simple as a single number, which scales up and down based on the desired number of servings. A measurement will be some base number, but will also need some notion of whether it's a volume or a weight or a count of things.

# Unit Testing

## Python unittest

Like `datetime` for working with dates and times, Python includes a standard library for writing unit tests. It's imaginitively called `unittest`. `unittest` exports a few utilities, but the main feature is a base class `TestCase`. Each test case we write will be a class that extends `TestCase`. These test case classes will use methods for each test case they want to execute. We'll write a test case class for each rug type, and within them write a test method for each of the edge cases we want to cover.

We'll write our first test in just a moment, but we do need to do a tiny bit of clean up and preparation. First, we want to write our test file in the same folder as the `rugs.py` program from chapter 3. We can either create `rugs_test.py` in the same folder as `rugs.py` lives now, or we can create a new folder, copy `rugs.py` there, and then create `rugs_test.py`. Whichever way you do it, we need to fix `rugs.py` a tiny little bit to make it work as a library.

Right now, `rugs.py` is a **script**, because it executes as a single file and immediately runs commands to start taking user input. While we want this behavior when we run the program, the unit test doesn't have user behavior. It just wants the class definitions from the file. But if we `import` the file right now, Python will find the class definitions, and then right away start asking whether the user wants to price a rug.

What we're going to do is put the last two lines (the `while` loop) in a function, and then add a couple lines of Python which will only execute that function when we run the file directly, as opposed to when it gets included as a library in another program.

```
def price_rugs():
    while not input("Price another rug (Y/n): ").lower().startswith("n"):
        price_rug()

if __name__ == "__main__":
    price_rugs()
```

This last part, `if name == "main":` takes advantage of a Python feature that lets us know if the file we're in is the file that we ran specifically, rather than a file which was included by another file. `name` is a special variable in Python, and when it has the value `"main"`, we know that the user wanted to run this file, rather than importing it.

Now we can write our first test case! Again, in `rugs_test.py`, add this code and run it!

```
import unittest
import rugs


class SquareRugTestCase(unittest.TestCase):
    def test_no_fringe(self):
        rug = rugs.SquareRug(5, False)

        area = rug.area()
        cost = rug.cost()

        self.assertEqual(area, 25)
        self.assertEqual(cost, 125)


if __name__ == "__main__":
    unittest.main()
```

When we run it, `unittest` has some great output for us:

```
.
----------------------------------------------------------------------
Ran 1 test in 1.236s

OK
```

This says that it took about a second to run one test, and all the tests pass!

The code to create the test starts by importing `unittest`, which we'll use to coordinate and organize the tests, and `rugs`, which is the name of the file which has our `Rug` classes. The test suite is `SquareRugTestCase`, which extends the `unittest.TestCase` utilities. These are responsible for triggering and gathering results, and reporting the final results at the end of the run. `SquareRugtestCase` has no constructor (`init` method) because it doesn't need to set up any contructors or do any initialization.

The first test, `test_no_fringe`, makes a square rug. It uses the values `5` and `False` for size and fringe. With the test set up, it moves on to evaluating the calculations. It saves `area`, `perimeter`, and `cost` in local variables. Finally, the assert uses the provided helper classes to check if the values are what we expect. `self.assertEquals(cost, 125)` compares the two values, and if they are not exactly equal, print and record the failing test.

To see a failing test, try changing one of the constants, and see what happens! Here's an example of mine:

```
F
======================================================================
FAIL: test_no_fringe (__main__.SquareRugTest)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "01_rug_tests\rug_test.py", line 14, in test_no_fringe
    self.assertEqual(perimeter, 25)
AssertionError: 20 != 25

----------------------------------------------------------------------
Ran 1 tests in 0.004s
```

So I changed the perimeter check to be 25, which is not the correct value. I can read this error message and see specifically the line that was at issue, and the problems associated. In appendix 3, we look in depth at reading and using exceptions to debug our programs.

Let's add another two tests!

```
class SquareRugTestCase(unittest.TestCase):
    def test_square_rug_no_fringe(self):
        # ... the test from before

    def test_square_rug_fringe(self):
        rug = rugs.SquareRug(5, True)

        area = rug.area()
        perimeter = rug.perimeter()
        cost = rug.cost()

        self.assertEqual(area, 25)
        self.assertEqual(cost, 155)

    def test_square_rug_no_size(self):
        rug = rugs.SquareRug(0, False)

        area = rug.area()
        perimeter = rug.perimeter()
        cost = rug.cost()

        self.assertEqual(area, 0)
        self.assertEqual(perimeter, 0)
        self.assertEqual(cost, 0)
```

And running this, it says we have three tests, OK!

These tests probably don't seem like a lot, but they give us quite a bit of easily verifiable information. While it looks like a long amount of code (and, it is...) the code is clear and direct in what it's doing. There's no logic, only direct calls and direct comparisons. While we've tried to be

concise and tight in our main program, being able to simply read the tests is a great boon.

## Writing your own tests

As you go to write your own unit tests, we need a way to take what we've done for rugs and apply it generally. The first thing we want to do is create a test file for every program file we create. In the rugs program, we had `rugs.py` and put the test in `rugs_test.py`. This convention, of creating a second file and adding `\_test` to it, is a great way to organize your project. It is clear what your program files are, and what the tests are.

The second thing to do is to make sure our program has no behavior when it's first called. That is, Python can run the file and find all the functions and classes and variables, but it won't execute any code right away. It needs to wait until it's told explicitly to run in a test or another file. But many of our programs do need to do something if we run them specifically! To achieve this, at the end of our python files we check if we're running the "main" file that was specified on the command line.

```
if __name__ == "__main__":
    # Run the program if it was called directly
```

Because we should be testing all our programs, any python file we write that does more than declare classes and functions should end with this block of code.

Third, we can import the pieces of our program into our test. Because the test file lives in the same folder as the program, we can just use the file name (without the .py extension) in the `from` part of our import. There are two ways we can import our implementation. Using the rugs example, we could use either of these forms:

```
import rugs

squareRug = rugs.SquareRug()
rectRug = rugs.RectangularRug()
```

This form imports the file as an object, where the properties are all the functions and classes at the top level of the `rugs.py` file. Alternatively, you can import each item in isolation. This is useful if you only need one or two items from the file.

```
from rugs import SquareRug
from rugs import RectangularRug

squareRug = SquareRug()
rectRug = RectangularRug()
```

We have used this form before when we only wanted one or two things out of a much larger module, like when we did `from math import sqrt`. While we're doing imports, we also want to import TestCase from the unittest module which Python provides.

After importing the parts of our program we intend to test, we can fourthly start creating TestCases. A TestCase does two things - it groups together tests for related pieces of functionality, and it extends from the Python TestCase base class to get a bunch of "magic" that will actually execute the tests when you run the test file.

To get that magic, we need to create a class structured in a very specific way. First, it must extend from unittest.TestCase. Second, it must have methods whose name begins with `test`. Python looks at classes which extend TestCase for any method that starts with `test`, and treats those as the specific tests to run and, whether they pass or fail, will report them with that name.

We can look at the Rugs tests for an example:

```
import unittest
import rugs

class RectangularRugsTest(TestCase):
    def test_rectangle_fringe(self):
        # The body of the test
```

When you create tests for your own projects, it's usually a good idea to have one test file per program file, and one test case in the test file per class or function in the program file. Each TestCase then has multiple tests methods to verify each part of functionality that it has. A simple function might have just a single test method, while a complex class could have dozens or hundreds!

Our fifth step in creating a test for our programs is writing out the details of each possible way to run the function or class. You want to be writing a test method for the common ways to use it, and you realy want to write a test method for each of the uncommon ways! For instance, in the rugs, we tested both with and without fringe. These are common cases. We also tested what happens if a size is 0! This and other uncommon cases are **edge cases**, and because they're dealing with things a bit wonky or out of the ordinary, we really want to focus on them in our tests to make sure it behaves just as we thing it should.

Within our test methods, there's a common approach: set up, execute, assert. It's easiest to look at these in backwards order. Assert is how we will verify our program is behaving correctly. We know that if it does perform as specified, our program will be in some certain state. As a very simple example, let's say our program is adding two numbers: `a = b + c`. After the program has completed this step, we expect the variable `a` to have a certain value. And to demonstrate that, we will `assert` that it does. We do this with the `assert⋯` methods. In rugs, this was `self.assertEqual(rug.cost, 50)`. For this example, it would be `self.assertEqual(a, the_value_of_b_plus_c)`.

| | |
|---|---|
| **NOTE** | We use the expected value of `b + c`, instead of calculating it inline! By manually performing the expected calculation, you save yourself from accidentally using a value that came from a bug in your code! And yes, most of the time the computation is much more complex than addition! |

`assertEqual` is the most common assertion you'll do, but others are available if you check the Python documentation.

To assert something about a program state, we must have executed it before. In the rugs test, that was calling the cost method and storing the result. In our simple example here, it would just be the one line of code `a = b + c`. For this to work, we need some values for `b` and `c`, and that's the set up part. All in all, this test would look like so:

```
# Create a test case
class ArithmeticTests(unittest.TestCase):
  # Declare a test method
  def test_simple_addition(self):
    # Set up
    b = 5
    c = 3

    # Execute
    a = add(b, c)

    # Assert
    self.assertEqual(a, 8)
```

There you go! These are all the pieces of defining, creating, and implementing a unit test for your program. The last piece is to run it. To do that, we're going to tell unittest to run when the test class is executed directly:

```
if __name__ == "__main__":
  unittest.main()
```

Putting that at the end of your `\_test.py` file will let you run it with python on the command line, and it will either tell you all the unit tests passed, or it will show you an error for exactly which one failed!

The whole example again:

```
import unittest # Get the testing library

from my_arithmetic import add # Get the thing we wrote and will test

# Create a test case
class ArithmeticTests(unittest.TestCase):
  # Declare a test method
  def test_simple_addition(self):
    # Set up
    b = 5
    c = 3

    # Execute
    a = add(b, c)

    # Assert
    self.assertEqual(a, 8)

# Run the tests when we execute this file
if __name__ == "__main__":
  unittest.main()
```

### Exercises

1. Write tests for the other rugs types you have.
2. Write tests for recipe ingredient scaling.

# Testing Maze

The maze program is much more complex than the rugs. It involves several objects that interact with one another and apply logic to evaluate and validation those interactions. For this test, we're going to start by thinking through our test suite before writing our test cases.

The maze has several classes, and we'll want one test case for each class. The idea will be to capture all the tests that are specific to a class in its own test case class, so there's a 1:1 correspondance. This doesn't mean we can't use other classes in these tests, just that the focus will be on one individual class at a time.

Let's start writing this down. Save a new file, `maze_test.py`. Make sure either you save it in the same folder as `maze.py`, or create a new folder and move `maze.py` so they're adjacent. In this file, start with the bare minimum of a unit test.

```
import unittest
import maze


# Test cases will go here


if __name__ == "__main__":
    unittest.main()
```

This imports unit test and maze, and if the file is executed directly, will run the unit tests.

Let's look at our test cases. We have `Room`, `Door`, `Lock`, `Keyring`, `Player`, and `Game`. Because `Game` deals with input and output, we'll cover its tests separately in the next section. Looking at the other five classes, we should start with classes who have the least dependencies on other classes, and write the simplest tests for that first.

`Lock` has no other classes that it needs for it to work. It just takes keys, which are strings, and checks if the key is in its list of keys that fit. Looking at what to test, I see four easily identifiable cases:

1. Can it add a key that fits at all?
2. Does it handle adding a duplicate key?
3. Will it accept a fitting key?
4. Will it reject a key that doesn't fit?

We can write all these out as individual test cases. But, I don't want to think about their implementation yet. I want us to take time to think about the test suite as a whole, before we look at writing individual tests. We can do that in Python by using the keyword `pass` as the body of a method - this tells Python that the method is really here, it just doesn't do anything (yet). We've stubbed it out.

```
class LockTestCase(unittest.TestCase):
    def test_add_fitting_key(self):
        pass

    def test_add_duplicate_fitting_key(self):
        pass

    def test_accepts_fitting_key(self):
        pass

    def test_does_not_accept_missing_key(self):
        pass
```

Here we have our `LockTestCase` which extends `TestCase`. It has four `test_` methods for each of the four cases. Each of those has the body `pass` which we can come back to later to fill in.

With the `Lock` tests sketched out, I think the `Keyring` makes sense. It's very similar to the `Lock`, but also needs to understand the difference between doors that are locked that it has the key for, doors that are locked that it doesn't have the key for, and doors that are simply unlocked.

```
class KeyringTestCase(unittest.TestCase):
    def test_add_key(self):
        pass

    def test_add_duplicate_key(self):
        pass

    def test_can_unlock_door_no_lock(self):
        pass

    def test_can_unlock_door_having_key(self):
        pass

    def test_cannot_unlock_door_missing_key(self):
        pass
```

We started using `Door`s in `KeyringTestCase`, so we might do their tests next. Most of the complexity around doors is actually in the lock. It's not particularly useful to test that a constructor did or did not set a property (if that's broken, we'd find out in other tests). So for the `Door`, we only need to test that the `other_side` function works.

```
class DoorTestCase(unittest.TestCase):
    def test_tracks_other_side(self):
        pass
```

Similar to the door and the lock, we won't test that a `Room` can correctly get a description. But we do need to test that a `Room` can add a door in each direction, that it adds keys to those doors, and that when a player first enters the room that it uses `on_enter` appropriately.

```
class RoomTestCase(unittest.TestCase):
    def test_add_north_door(self):
        pass

    # Three other directions, for you to stub out

    def test_on_enter_adds_key(self):
        pass
```

As we said, we're leaving the `Game` tests to the next section, so the only one we have left is the `Player`. All the Player logic is around the `move_through` method, so we'll write a test for each of these cases.

```
class PlayerTestCase(unittest.TestCase):
    def test_can_move_through_unlocked_door(self):
        pass

    def test_can_move_through_locked_door_with_key(self):
        pass

    def test_cannot_move_through_locked_door_without_key(self):
        pass

    def test_cannot_move_through_None_door(self):
        pass
```

Let's take a moment to look over these classes. We have tests for each of the individual pieces of logic. With this level of coverage, we can be pretty confident that our code will behave as expected, if each of these tests pass. All we need to do is actually write them!

```
class LockTestCase(unittest.TestCase):
    def test_add_fitting_key(self):
        lock = maze.Lock()

        lock.add_fitting_key("blue")

        self.assertEqual(lock.fitting_keys, ["blue"])

    def test_add_duplicate_fitting_key(self):
        lock = maze.Lock()

        lock.add_fitting_key("blue")
        lock.add_fitting_key("blue")

        self.assertEqual(lock.fitting_keys, ["blue"])

    def test_accepts_fitting_key(self):
        lock = maze.Lock()
        lock.add_fitting_key("blue")

        accepts = lock.accepts("blue")

        self.assertTrue(accepts)

    def test_does_not_accept_missing_key(self):
        lock = maze.Lock()
        lock.add_fitting_key("blue")

        accepts = lock.accepts("red")

        self.assertFalse(accepts)
```

```
class KeyringTestCase(unittest.TestCase):
    def test_can_unlock_door_no_lock(self):
        keyring = maze.Keyring()
        door = maze.Door(maze.Room(), maze.Room())

        can_unlock = keyring.can_unlock(door)

        self.assertTrue(can_unlock)

    def test_can_unlock_door_having_key(self):
        key = "blue"
        keyring = maze.Keyring()
        lock = maze.Lock()
        keyring.add_key(key)
        lock.add_fitting_key(key)
        door = maze.Door(maze.Room(), maze.Room(), lock)

        can_unlock = keyring.can_unlock(door)

        self.assertTrue(can_unlock)
```

```
class DoorTestCase(unittest.TestCase):
    def test_tracks_other_side(self):
        room_a = maze.Room("A")
        room_b = maze.Room("B")
        door = maze.Door(room_a, room_b)

        other_size_a = door.other_side(room_a)
        other_size_b = door.other_side(room_b)

        self.assertEqual(other_size_a, room_b)
        self.assertEqual(other_size_b, room_a)
```

```
class RoomTestCase(unittest.TestCase):
    def test_add_north_door(self):
        room = maze.Room("A")
        north_room = maze.Room("B")

        room.add_north_door(north_room)

        self.assertEqual(room.north_door.other_side(room), north_room)
        self.assertEqual(
            north_room.south_door.other_side(north_room), room)
        self.assertIsNone(north_room.north_door)

    def test_on_enter_adds_key(self):
        room = maze.Room("A", "blue")
        player = maze.Player()

        room.on_enter(player)

        self.assertIn("blue", player.keyring.keys)
```

```
class PlayerTestCase(unittest.TestCase):
    def test_can_move_through_locked_door_with_key(self):
        key = "blue"
        room_a = maze.Room("A")
        room_b = maze.Room("B")
        lock = maze.Lock()
        lock.add_fitting_key(key)
        room_a.add_north_door(room_b, lock)
        player = maze.Player(room_a)
        player.keyring.add_key(key)

        player.move_through(room_a.north_door)

        self.assertEqual(player.current_room, room_b)
```

# Advanced Topics

```
class GameTestCase(unittest.TestCase):
    def setUp(self):
        self.game = maze.basic_maze()

    @unittest.mock.patch("builtins.print")
    def test_print_room(self, mock_print):
        self.game.print_room()

        mock_print.assert_any_call("You are in the entry way.")
        mock_print.assert_any_call("There is a door to the (n)orth.")
        mock_print.assert_any_call("There is a door to the (e)ast.")

    def test_move_player(self):
        kitchen = self.game.end_room
        dining_room = kitchen.south_door.other_side(kitchen)

        self.game.move_player("n")

        self.assertEqual(self.game.player.current_room, dining_room)

    @unittest.mock.patch("builtins.print")
    def test_move_player_no_room(self, mock_print):
        self.game.move_player("s")

        self.assertEqual(
            self.game.player.current_room, self.game.start_room)

    @unittest.mock.patch("builtins.print")
    @unittest.mock.patch("builtins.input")
    def skip_test_play_stuck(self, mock_print, mock_input):
        mock_input.side_effects = ["n", "n", "q"]

        self.game.play()

        mock_print.assert_any_call("You are in the entry way.")
        mock_print.assert_any_call("You are in the library.")
        mock_print.assert_any_call("Cannot move through that door.")

    @unittest.mock.patch("builtins.print")
    @unittest.mock.patch("builtins.input")
    def skip_test_play_win(self, mock_print, mock_input):
        mock_input.side_effects = ["e", "w", "n", "n"]

        self.game.play()

        mock_print.assert_any_call("Picked up blue key")
        mock_print.assert_any_call("You are in the library.")
        mock_print.assert_any_call("You are in the dining room.")
        mock_print.assert_any_call("You won the treasure")
```