Software Craftsmanship

Python Workbook

David Souther

2020-11-27

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: Python

Python 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

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