File: worksheet

Life Conference Class - Introduction to Programming

February 1, 2019

Andrew Gallant
jamslam@gmail.com

Kaitlyn Brady
kbrady39@gmail.com

Vocabulary

If you don't understand all of these at first, that's OK! We'll see examples of them throughout the class.

programming:

> The process of designing and building a program for solving a specific problem.

program:

> A specific sequence of instructions that precisely instruct a computer to do something. Software consists of one of more programs.
>
> Programs are normal files on your computer, just like anything else. The only difference between a program file and other files is that a program file can be turned into a process.

process:

> A single instance of a running program on your computer.

programming language:

> A set of instructions with which to write a program. Source code is made up of these instructions, and the specific way in which one writes the instructions is called the programming language's syntax.

text editor:

> A program that one uses to manipulate the source code for a program.

shell:

> A tool for issuing commands directly to the computer.

Getting Started

For today, we will be working exclusively in your computer's shell. To open a shell on macOS, do the following:

1. Click on a blank space in the desktop.
2. Make sure it says "Finder" in the top left corner.
3. Click "Go" and select "Utilities" from the menu.
4. Double click on the "Terminal" icon. A shell will open.
5. Move and resize the shell so that it uses the left half of your screen.

6. Open a second shell by clicking "Shell" near the top left corner, and then select "New Window" from the menu. A second shell will now open.
7. Move and resize the second shell so that it uses the right half of your screen.

At this point, you can now issue commands to your computer! Test it out by typing "ls" (without the quotes) and pressing enter. What happens? Type in something else (anything) and press enter. What happens?

Tip: use Command+` to switch between shells quickly using only the keyboard.

Basic commands

In order to build our spell checker, we'll need some text to spell check and a dictionary of words. Let's setup a directory for our project and download these files.

```
$ mkdir spell-checker
$ cd spell-checker
$ curl -O https://burntsushi.net/stuff/life-conference/dictionary
$ curl -O https://burntsushi.net/stuff/life-conference/text
```

If you run

```
$ ls
```

You should see two files in your current directory: dictionary and text.

So what do these commands do?

ls - List the contents of the current directory.

cd - Change the current directory.

mkdir - Create a new directory inside the current directory.

curl - Download a file from the Internet.

Let's explore the data files we downloaded by learning about more commands:

cat - Print contents of a file.

wc - Count the words in a file.

Creating our first program

In order to write a program, we first need to learn how to use a text editor. There are many different text editors to choose from. Some of them are very advanced and actually help you write programs. In this class, however, we will be using a very simple text editor called nano. You can run it right from your shell:

```
$ nano
```

Once you're in nano, you can start typing your program. So let's write our first one. Type in the following:

```
print "Hello, world!"
```

Once you've typed in the program, you now need to write the program to a file. You can do that by pressing Control+O. You will be asked to give the name of the file you want to write. Type "program.py" (without the quotes) and press enter. Your program is now saved!

Now all we have to do is run the program. Switch to your other shell, and change into the directory you created before:

```
$ cd spell-checker
```

and now list the contents of the directory:

```
$ ls
```

In addition to the data files you downloaded previously, you should now see a new file "program.py". This means the program has been saved as a file and is ready to run. To run the program, use the following command:

```
$ python program.py
```

What happens?

Syntax

When telling a computer what to do, it is very important to be precise. One small slip-up or one small typo can cause your program to fail. This may happen to you a lot. The process of figuring out what's wrong with your program and how to fix it is called debugging, and it is something that programmers spend a lot of time doing.

Let's see an example of how a program can misbehave. Switch back over to the shell that contains the program you wrote in the previous step. Once there, change "print" to "pint". Save the file by pressing Control+O and hitting enter (you do not need to type the file name again). Now switch back over to your other shell and run your program again:

```
$ python program.py
```

What happens? Experiment with other changes to the program. What is allowed and what isn't?

Variables

Variables are an important aspect of almost all programs. Variables allow you to associate values with a name, and then use that name to reference that value later. This is best seen with example. Go back to your shell with source code in it, and change the code to use a variable.

```
message = "Hello, world!"
print message
```

Now save the file and run the program. What do you see?

In this example, "message" is a variable. We store the string "Hello, world!" in the variable, and then print the variable in the next line.

What happens if you change the variable name? What about the contents of the

variable?

If statements

"if" statements are another important aspect of almost all programs. An "if" statement lets you execute something only if a particular condition is true or not. An example is helpful here. Go back to your shell with source code in it, and change the code to this:

```
message = "abc"
if len(message) > 3:
    print message
```

Before running the program, try to guess what it will print. Now try running the program. Were you right?

What do you think will happen if you change "3" to "2"? Try doing it to see whether you guessed right!

Loops

Loops are a way to execute something more than once without having to explicitly repeat it. For example, let's say we wanted to print a message 5 times along with the message's number. You could do this:

```
message = "Hello, world!"
print 1, message
print 2, message
print 3, message
print 4, message
print 5, message
```

In this program, we assign our message to a variable, and then print that message 5 times along with its number. With a loop, we can avoid repeating ourselves! For example:

```
message = "Hello, world!"
for i in range(5):
    print i, message
```

Does this program print the same thing as the previous program? Try it. What happens if you change "5" to something else?

Reading files

Before writing a spell checker, we need to first learn how to read data from other files. Let's try the simplest possible thing: read data from a file and then print it back out again. (Does this sound familiar? This is what the "cat" program does!)

Here's a program that does just that:

```
for line in open("text"):
    print line
```

In this program, we use 'open("text")' to open the file called "text" (which we downloaded at the beginning of class). We then use a loop to execute some

code for every line in the "text" file.

Compare the output of your program with the output of

```
$ cat text
```

What's different? How can we fix it?

Hint: line.strip("\n") will give you a line without the end of line terminator.

Counting words

There's one last thing we need to learn before we can write a spell checker, and that's how to look at words. To start with, let's try writing a version of the "wc" program that counts the total number of words in a file. Start by writing out an algorithm:

1. Create a count variable set to 0.
2. Loop over all lines in text.
3. Loop over all words in the line.
4. Add 1 to the count variable.
5. After looping over the lines, print the count variable.

Here's the code for the algorithm above:

```
count = 0
for line in open("text"):
    line = line.strip("\n")
    for word in line.split():
        count += 1
print count
```

In this program, we loop over all of the lines in a file. We then use "line.split()" to loop over all words in each line. Finally, we increment a counter by 1 every time we see a word. Does this match the output of the "wc" command shown below?

```
$ wc -w text
```

Here's a twist. What if instead of counting the total number of words, we wanted to count the total number of unique words?

Let's try to sketch out an algorithm for this:

1. Create an empty set.
2. Loop over all lines in text.
3. Loop over all words in the line.
4. Add the word to the set.
5. After looping over the lines, print the size of the set.

Now let's translate the above algorithm into code:

```
words = set()
for line in open("text"):
    line = line.strip("\n")
    for word in line.split():
        words.add(word)
print len(words)
```

Can you think of a problem with the above approach? Are "school" and "School" the same word or different words?

Building a spell checker

Finally, you have everything you need to build a spell checker. As with previous examples, let's try to sketch out an algorithm for our spell checker:

1. Create a set of words from a dictionary.
   1a. Initialize an empty set of words.
   1b. Loop over each line in "dictionary".
   1c. Add each line to the set.
2. Check if there are any words in "text" that aren't in the dictionary.
   1a. Loop over all the lines in "text".
   1b. Loop over all the words in each line.
   1c. For every word, check whether it's in the dictionary.
   1d. If the word isn't in the dictionary, print it.

Hint: To check whether a word is in a set or not, use this:

```
if word not in dictionary:
```

Here's the final code you should roughly end up with, including lowercasing and stripping punctuation:

```
dictionary = set()
for word in open("dictionary"):
    word = word.strip("\n").lower()
    dictionary.add(word)

for line in open("text"):
    line = line.strip("\n")
    for word in line.split():
        word = word.lower().strip(",.")
        if word not in dictionary:
            print word
```