Notebook

March 30, 2019

1 Introduction to Programming and Python

In this first week of the course, we're going to cover some core concepts of computer programming. With this, we will build a foundation to do more interesting things in the coming weeks.

1.1 What is Computer Programming?

Computers basically just do one thing: they complete very simple operations at astonishing speed.

For example, suppose you wanted to calculate the thousandth number of the Fibonacci Sequence — the sequence of numbers such that starts with 0 and 1, and then each subsequent number is the sum of the previous two numbers. For example:

0 1 1 2 3 5 8 13

Don't go on until you are satisfied you know what the next number is.

As you can imagine, it would take a fairly long time to calculate it by hand. However, a computer can do it in less than a second. I'll show you:

```
In [1]: digit = 3
        first = 0
        second = 1
        while digit < 1001:
            answer = first + second
            first = second
            second = answer
            digit += 1
        print("the " + str(digit - 1) + "th number in the Fibonacci sequence is: " + str(answer)
```

```
the 1000th number in the Fibonacci sequence is: 2686381002448535938614672720214292396761660931898
```

This is obviously not a number you would want to try to compute by hand, or even using a pocket calculator! (Incidentally, if you looked up a list of those numbers and found a different result, try comparing it to the 999th number. People seem to differ on whether to start counting from 0 or 1 in enumerating the Fibonacci sequence.)

Everything that computers do—including the system you're reading this on right now—is built up by doing very simple ("primitive") operations many many times. The way it works is that programmers build those primitives—basic arithmetic operations and pushing things around in memory, basically—into more complicated pieces of functionality. Then the next programmer can use that complicated piece of functionality (often called an "abstraction") and build some of those up into something more complicated, and so on, and so on.

We will start at a much higher level, but you'll learn to do that yourself!

Let's start by breaking down the Fibonacci sequence code above:

```
digit = 3
first = 0
second = 1
while digit < 1001:
    answer = first + second
    first = second
    second = answer
    digit += 1
print("the " + str(digit - 1) + "th number in the Fibonacci sequence is: " + str(answer))
```

The first three lines are **variable assignments** — they give a name to some piece of data. In this case, we've given the name `digit` to the number 3, and so forth. Once we give a name to some piece of data, we can refer to that name and get back that data—and we can change the data attached to the name as well.

It's good to use descriptive names, so anyone else reading your code (or future you) can tell what you're trying to do. Here, I used names meant to track exactly what we're doing: the `digit` starts at 3 because we start calculating at the third digit of the Fibonacci sequence (we know the first two already). Likewise, `first` and `second` represent the two numbers we need to add up to get the third.

`while digit < 1001:`

The 4th line starts a **while loop**. Remember how I said that computers are really good at doing the same thing over and over? The most basic mechanism for doing so is the loop, which is just programmer-speak for "execute a list of instructions repeatedly."

The structure ("syntax") of the while loop in Python (it's different in other languages) is:

1. The keyword `while` at the start of the line.

2. An *expression* evaluating to a *boolean* after that. (I'll explain what the italicized terms mean in a moment.) This is known as the condition.

3. A colon at the end of the line.

4. An indented series of lines ("block") containing *statements*, known as the "body" of the loop.

The way it works is that each of the statements in the indented block will be executed in sequence so long as the condition is true.

Thus, the English translation of the Python `while digit < 1001:` is "go look up the value of the `digit` variable. So long as it is less than 1001, execute all the indented statements that follow."

Let's talk about some of those vocabulary words.

First, most of what you can write in Python can be divided into expressions and statements. Essentially (and I'm simplifying a bit here), an **expression** is like algebra: it's a representation of some value, and its value can be substituted in for it. For example `1 + 1` is an expression, and 2 can seamlessly be substituted in for it. We say that an expression **evaluates** to its value: `1 + 1` evaluates to 2.

In `while digit < 1001` the `digit < 1001` part is an expression. It evaluates to a special type of result, known as a **boolean**. In Python, the two boolean values are **True** and **False** and they mean pretty much what you would expect them to mean. So `digit < 1001` tells the computer “go look up the value of `digit` and see if it’s less than 1001. If so, replace the value of this expression by `True`; otherwise, replace it with `False`.

Statements can be seen as instructions to the computer. They aren’t meant to be evaluated to some value, but are meant to bring something about in the world inside the computer’s head. We’ve seen a lot of statements already: the variable assignments we’ve looked at are all statements. They don’t calculate anything, they just tell the computer “hold onto this name for this quantity.”

If we wrote a line like `my_sum = 1 + 1` we would have an expression and a statement. What happens is that first the expression `1 + 1` is evaluated to 2, and then that value is assigned to the variable `my_sum`. We always read variable assignments right to left: evaluate all the stuff on the right side of the equals sign, and then assign whatever we get out of it to the name on the left side.

Generally, you can have a lot of expressions in a line — you could say something like `weird_quantity = 5 * (9 + 28) * 2015 / 99` and that’s fine. Generally, you don’t have more than one statement in a line.

Sometimes, expressions do something (like statements do) in addition to evaluating to a value. Those are called *side-effects* but don’t worry about that for now.

Ok, let’s keep going! You should be able to figure out what `answer = first + second` means— it assigns the variable `answer` to the expression `first + second`. It looks up the values of `first` and `second` and sums them. You should see that `answer` get the value 1.

The next two lines introduce one of the main reasons we use variables. As the name suggests, it’s because they can *vary*.

```
first = second
second = answer
```

We’re reassigning our variables here! Remember, we do variable assignment right to left. So what this means is “take the value currently attached to the name `second` and assign it to `first`, then take the variable currently attached to the name `answer` and assign it to `second`.

Think about how this works in terms of the first few digits of the Fibonacci sequence. Consider the following table, where the first row is the digit we’re on (first, second, etc.) and the second row is the value:

		/				
1	2	3	4	5	6	7
0	1	1	2	3	5	8
ˆ1	ˆ2					

Let’s imagine an arrow poing to the digit we’re trying to sort out. So when we start, the arrow is pointing at 3 (the variable `digit`), because we’re trying to calculate the third digit. We calculate that digit by looking to the first number of the two immediately preceeding numbers (the variable `first`, represented in the table by ˆ1), and adding it to the second number of the two immediately preceeding (the variable `second`, represented in the table by ˆ2), giving us the total (variable `answer`).

But we’re not done yet, because we still have 997 more numbers to calculate. So how do we calculate the 4th digit? Well, we need to add the last two digits again. . . the number that used to be in the variable `second` is the first digit we'll add the next time, and the current digit (variable `answer`) is the second one. In order to tell the computer that's going to happen, we reassign those variables.

The last line of our loop `digit += 1` is a special syntax for *incrementing* the value of a variable—it just says "replace the value of `digit` with its own value plus 1." In our chart above, that just means moving the digit forward.

So, here's a representation of out computer's memory at the start of the indented block:

		/
1	2	3
0	1	
ˆ1	ˆ2	

And here's a representation of out computer's memory at the end of the indented block:

			/
1	2	3	4
0	1	1	
	ˆ1	ˆ2	

The arrow indicating what digit we're about to calculate has moved to the 4th place, and the arrows indicating what digits we're going to sum up have moved to the second and third, plus we've added an answer for the latter.

And finally, the magic of loops kicks in. Now that we're at the end of our indented block, the computer jumps back up to the start of the while loop.

First, it checks the condition again. Is `digit` still less than 1001? Yes, it is. It's 4 now, instead of 3, but that's lots less than 1001. So now, we go back into the body of the loop, this time using the new values of our variables!

After another iteration of the loop, here's what the computer knows:

				/
1	2	3	4	5
0	1	1	2	
		ˆ1	ˆ2	

And then it keeps doing that, again and again, moving the arrow up top each time, at blinding speed.

Now let's think of what happens after it calculates the 1000th digit. Here's where we are:

								/
1	2	3	4	5	…	999	1000	1001
0	1	1	2	3	…	lots	lots	
					…	ˆ1	ˆ2	

This time, when it goes back to the start of the loop, it tests the condition and finds that it's `False`. `digit` isn't under 1001 any more!

So now, it jumps out of the while loop and goes to the next line:

```
print("the " + str(digit - 1) + "th number in the Fibonacci sequence is: " +
str(answer))
```

This line calls the `print()` function, which instructs the computer to print to the screen (not to a paper printer!) the expression inside the parentheses. (We'll talk about functions and what the parentheses do in a little bit.)

The stuff inside the parentheses is a **string**—that is, an piece of text that the computer represents in memory, made up of **characters**. Typically we tell Python to hold onto a string by surrounding it with quotes (single or double both work, but they have to match). These are called **string literals**. Like:

```
my_sentence = "Hi, I am Gowder's sentence."
```

But you can also create strings out of expressions, and use them in other expressions. One of the neat things that you can do with strings is **concatenate** them, that is, use the plus sign to sort of add them together, so that `"the cat " + "in the hat"` evaluates to `"the cat in the hat"`.

So what that last line does is look up the last value that `answer` got (which is the 1000th digit, because we haven't calculated the 1001th yet), convert that into a string with the `str()` function (turning it from, e.g., `12345` into `"12345"`—this matters!), does the same with `digit`, and then concatenates them all together with the string literals there into one big string, which we print to the screen so that we can learn the answer to our math puzzle!

1.2 Now you try!

We're at the end of our first handout. There are a couple more in this lesson, but, before you go onto the next one, try using a loop to calculate a different quantity!

As you might remember from your early schooling, the factorial of a number, denoted n!, is the product of that number and all the positive integers before it. For example, 5! = 1x2x3x4x5 = 120

In the next cell, use a loop to calculate 1000 factorial.

```
In [ ]:
```