0 Meet Python

One way that you can spot a computer scientist is that they begin counting from 0 rather than from 1. So this is Chapter 0. But it's also Chapter 0 to signify that it's a warm-up chapter to get you on the path to feeling comfortable with Python, the programming language that we'll be using in this book. Every subsequent chapter will begin with an application in biology followed by the computer science ideas that we'll need to solve that problem.

Python is a programming language that, according to its designers, aims to combine "remarkable power with very clear syntax." Indeed, Python programs tend to be relatively short and easy to read. Perhaps for this reason, Python is growing rapidly in popularity among computer scientists, biologists, and others.

The best way to learn to program is to experiment! Therefore, **we strongly urge you to pause frequently as you read this book and try some of the things that we're doing here (and experiment with variations) in Python.** It will make the reading more fun and meaningful.

This chapter includes a number of short exercises that we encourage you to try. Subsequent chapters have links to end-of-chapter programming problems.

The link below offers instructions on how to install and run Python on your computer.

`www.cs.hmc.edu/CFB/Setup`

If you're reading this book as a part of a course, your instructor may have some additional or different instructions.

0.1 Getting Started

When you start up Python, you'll see a "prompt" that looks like this:

```
>>>
```

You can type commands at the prompt and then press the "Return" (or "Enter") key and Python will interpret what we've typed.

For example, below we've typed `3 + 5` (throughout this book, the user's input is always shown in black and Python's response is always shown in blue).

```
>>> 3 + 5
8
```

Next, we can do fancier things like this:

```
>>> (3 + 5) * 2 - 1
15
```

Notice that parentheses were used here to control the order of operations. Normally, multiplication and division have higher *precedence* than addition and subtraction, meaning that Python does multiplications and divisions first and addition and subtractions afterwards. So without parentheses we would have gotten...

```
>>> 3 + 5 * 2 - 1
12
```

You can always use parentheses to specify the order of operations that you desire.

Here are a few more examples of arithmetic in Python:

```
>>> 6 / 2
3
>>> 2 ** 5
32
>>> 10 ** 3
1000
>>> 52 % 10
2
```

You may have inferred what `/`, `**`, and `%` do. Arithmetic symbols like `+`, `-`, `/`, `*`, `**`, and `%` are called *operators*. The `%` operator is pronounced "mod" and it gives the remainder that results when the first number is divided by the second one.

0.2 Big Numbers

Python is not intimidated by big numbers. For example:

```
>>> 2 ** 100
1267650600228229401496703205376L
```

The "L" at the end of that number stands for "Long." It's just Python's way of telling you that the number is big. Look at this:

```
>>> (2 ** 100) % 3
1L
```

The number 1 is *not* long, but `2 ** 100` was long; once Python sees a mathematical expression with a long number in it, it stays in the long number state of mind. You don't have to worry about that – we just note this so you won't say "Huh!?" when you see the "L."

0.3 Strange Division

If you're using Python version 3, division works as you'd expect.

```
>>> 5 / 2
2.5
```

If you're using Python version 2 (e.g., version 2.7), the way it works may surprise you. Take a look at this:

```
>>> 5 / 2
2
```

We've consulted with expert mathematicians and they've verified that 5 divided by 2 is not 2! What's wrong with Python!? The answer is that Python (version 2) is doing "integer division". It assumes that since 5 and 2 are integers, you are only interested in integers. So when it divides 5 by 2, it rounds down to the nearest integer, which is 2. If you wanted Python to do "decimal division", you could do this:

```
>>> 5.0 / 2.0
2.5
```

Since we typed 5.0 and 2.0, Python realized that we are interested in numbers with decimal points, not just integers, so it gave us the answer with a decimal point. In fact, if just ONE of the 5 or 2 has a decimal point after it, Python will get the message that we are thinking about decimal numbers. So we could do any of these:

```
>>> 5.0 / 2
2.5
>>> 5 / 2.0
2.5
>>> 5. / 2
2.5
>>> 5 / 2.
2.5
```

0.4 Naming Things

Python lets you give names to values. This is very useful because it allows you to compute a value, give it a name, and then use it by name in the future. Here's an example:

```
>>> myNumber = 42
>>> myNumber * (10 ** 2)
4200
```

In the first line, we defined `myNumber` to be 42. In the second line, we used that value as part of a computation. The name `myNumber` is called a *variable* in computer science. It's simply a name that we've made up and we can assign values to it.

Notice that the "=" sign is used to make an assignment. On the left of the equal sign is the name of the variable. On the right of the equal sign is an expression that Python evaluates, assigning the resulting value to the variable. For example, we can do something like this:

```
>>> pi = 3.1415926
>>> area = pi * (10 ** 2)
>>> area
314.15926
```

In this case, we define the value of `pi` in the first line. In the second line, the expression `pi * (10 ** 2)` is evaluated (its value is 314.15926) and that value is assigned to another variable called `area`. Finally, when we type `area`, Python displays the value. Notice, also, that the parentheses aren't actually necessary here. However, we used them just to help remind us which operations will be done first. It's often a good idea to do things like this to make your code more readable to other humans.

One last thing before we move on. In mathematics, the expression `pi = 3.1415926` is equivalent to the expression `3.1415926 = pi`. In Python, and in almost all programming languages, *these are not the same*! In fact, while Python understands `pi = 3.1415926` it will complain loudly if you type `3.1415926 = pi`. (Try it in Python to see what it looks like when Python complains.) Didn't Python go to elementary school? The answer is that programming languages view the = symbol in a special way. They assume that what's on the left-hand side is the name of a variable (`pi` in our example). On the right-hand side of = is an expression that can be evaluated. That expression is evaluated and its value is then associated with the variable.

So, when Python sees...

```
>>> pi = 3.1415926
```

... it evaluates the expression on the right. That's easy, there's nothing to evaluate – it's 3.1415926. Python then assigns that value to the variable `pi` on the left. Now, when Python sees...

```
>>> area = pi * (10 ** 2)
```

... it evaluates the expression on the right. That's not too hard either; we've already defined `pi` to be 3.1415926 and Python does the math and evaluates the expression on the right to be 314.15926. It then associates that value with the variable named `area`.

Sometimes, you'll see (and write) things like this:

```
>>> counter = 0
>>> counter = counter + 1
```

In the first line, we've set a variable named `counter` to 0. The second line makes no sense from a mathematical perspective, but it makes complete sense based on

what we now know about Python. Python begins by evaluating the expression on the right side of the = sign. Since `counter` is currently 0, that expression `counter + 1` evaluates to 1. Only once this evaluation is done, Python sets the variable on the left side of the = sign to be this value. So now `counter` is set to 1. This is how we'll often count the number of events of some sort – like the number of occurrences of a particular nucleotide in a DNA sequence or the number of species with a particular property.

0.5 What's in a Name?

Variable names are pretty much up to you. We didn't have to use the names `myNumber`, `pi`, and `area` in our examples above. We could have called them `Joe`, `Sally`, and `Melissa` if we preferred. Variable names must begin with a letter and there are certain symbols that aren't permitted in a variable name. For example, naming a variable `Joe` or `Joe42` is fine, but naming it `Joe+Sally` is not permitted. You can probably imagine why. If Python sees `Joe+Sally` it will think that you are trying to add the values of two variables `Joe` and `Sally`. Similarly, there are a few built-in Python "special words" that can't be used as a variable name. If you try to use them, Python will give you an *error message*. (An error message is the technical term for a complaint.)

Finally, it's a good practice to use descriptive variable names to help the reader understand your program. For example, if a variable is going to store the area of a circle, calling that variable `area` is a much better choice than calling it something like `z` or `y42b` or `harriet`.

Variables provide a convenient way to store values and the names that we give to variables should be simple and descriptive.

0.6 From Numbers to Strings…

That was all good, but numbers are not the only useful kind of data. Python allows us to define strings as well. A *string is* any sequence of symbols within either single or double quotation marks. Here are some examples:

```
>>> name1 = "Ben"
>>> name2 = 'Jerry'
```

```
>>> name1
'Ben'
>>> name2
'Jerry'
```

Notice that in the first example, we used double quotes around the string `"Ben"` and single quotes around the string `'Jerry'`. Python is happy either way – these are both fine for defining strings. However, notice that Python's own responses (in blue above) use single quotes even if we defined them with double quotes (as in the case of `"Ben"`). That's not a very important detail, but we mention it to avoid any confusion.

A string is any sequence of characters (numbers, letters, and other symbols) beginning and ending with either single or double quotation marks. If the string begins with a single quotation mark it should end with a single quotation mark and if it begins with a double quotation mark it should end with a double quotation mark.

In biology, the term "sequence" generally refers to a sequence of nucleotides or amino acids. Biological sequences are naturally represented by Python strings. Throughout this book, we'll use the terms *string* and *sequence* interchangeably, even though technically one is a computer science concept and the other is a biology concept.

Finding the length of a string

While there are many things that we can do with strings, here are a few of the most important ones. First, we can find the length of a string by using Python's `len` function:

```
>>> len(name1)
3
>>> len('hi there')
8
```

In the first example, `name1` (which we defined above) is `"Ben"` and the length of `"Ben"` is 3. In the second example, `'hi there'` contains a space (which is a regular symbol) so the total length is 8.

Indexing

Another thing that we can do with strings is find the symbol at any given position or *index*. The first symbol in a string actually has index 0 in Python. (As we noted at the beginning of this chapter, computer scientists like to start counting from 0.) So,

```
>>> name1[0]
'B'
>>> name1[1]
'e'
>> name1[2]
'n'
>>> name1[3]
IndexError: string index out of range
```

Notice that although `name1`, which is `"Ben"`, has length 3, the symbols are at indices 0, 1, and 2 by the way that Python counts. There is no symbol at index 3, which is why we got an error message.

0.7 Slicing

The next thing we want to show you is called *slicing*. Python lets you find parts of a string using its special slicing notation. Let's look at some examples first:

```
>>> goodFood = 'chocolate'
>>> goodFood[0:3]
'cho'
>>> goodFood[0:4]
'choc'
```

What's going on here? `goodFood` is a variable representing the string `'chocolate'`. The name of our variable is our own invention – Python has no prior notion of `goodFood` or what it might represent. Now, `goodFood[0:3]` is telling Python to look at the thing which `goodFood` represents (the string `'chocolate'`) and give us back the part beginning at index 0 and going up to, but not including, index 3. So, we get the part of the string with symbols at indices 0, 1, and 2, which are the three letters `'c'`, `'h'`, and `'o'` or `'cho'`.

It may seem strange that the last index is not used – so we don't actually get the symbol at index 3 when we ask for the slice `goodFood[0:3]`. It turns out that there are some good reasons why the designers of Python chose to do this. Humor us for now and we promise to revisit this later.

We don't have to start at 0. For example:

```
>>> goodFood[3:7]
'cola'
```

This is giving us the slice of `'chocolate'` from index 3 up to, but not including, index 7.

We can also do things like this:

```
>>> goodFood[1:]
'hocolate'
```

When we leave the number after the colon blank, it just assumes we mean "go until the end." So, this is just saying, "give me the slice that begins at index 1 and goes to the end." Similarly,

```
>>> goodFood[:4]
'choc'
```

Because there was nothing before the colon, it assumes that we meant 0. So this is the same as `goodFood[0:4]`.

Indexing and slicing allow us to select regions of interest in a string.

0.8 Adding Strings

Strings can also be added together. Adding two strings results in a new string that is simply the first one followed by the second one. This is called *concatenation* of strings. For example:

```
>>> 'yum' + 'my'
'yummy'
```

Once we can add, we can also multiply!

```
>>> 'yum' * 3
'yumyumyum'
```

In other words, `'yum' * 3` really means `'yum' + 'yum' + 'yum'`, which is `'yumyumyum'`; the concatenation of `'yum'` three times.

0.9 Negative Indices

Finally, notice that while it's easy to get the first symbol in a string, it requires a bit more work to get the last symbol. For example, after defining the string `goodFood = 'chocolate'`

we could get the last symbol this way:

```
>>> goodFood[len(goodFood)-1]
'e'
```

This works because the length of the string minus one is the last index of the string, but it's a cumbersome way to index that position! So, Python allows us to get the last symbol in the string using index –1, as in `goodFood[-1]`. Similarly, the second to the last symbol is at `goodFood[-2]`, and so forth.

You can use negative numbers in slicing as well. For example, to get the part of `mystring` that excludes the first and last symbols we could do this:

```
>>> goodFood[1:-1]
'hocolat'
```

0.10 Fancy Slicing

Python's slicing notation also allows you to do things like this:

```
>>> mystring = 'abcdefghijklmnop'
>>> mystring[1:9:3]
'beh'
```

What happened here? The 1 and 9 here mean that we want the slice from index 1 to index 9. In this case that's the string `'bcdefghi'`. The last number, 3, means that we just want every third symbol in that string. So, that is `'b'` (now skip three symbols forward) then `'e'` (now skip three symbols forward) and then `'h'`. There is no symbol 3 symbols forward from `'h'` in the slice `'bcdefghi'`.