1 Administrative Topics

- Any questions about the projects?

2 Python and its memory model

To learn how to program effectively in Python, we must understand how the language is storing and manipulating data – we must understand its “memory model.”

I think of Python programs as a sequence of commands that manipulate data in memory.

To learn Python, we need to learn two things:

1. What are the legal Python commands?
2. How does Python view memory? I.e. what is Python’s memory model?

In the interpreter, I can show you a few Python commands:

- printing something to the screen: (`print 'hello'`)
- adding two numbers: (`3+4`)
• printing the result of a computation (print "The sum is ", 3+4)
• adding more numbers: (42+0.1)

That’s great, I can use Python like a calculator – I can type commands and it will do things. But what if I want to keep track of results from earlier computations? The answer is that I want to label the data. I want to create a named variable and associate some particular value with it. I do that with an assignment statement.

a = 3 + 4

This line of code is calculating the sum of 3 and 4, then placing that result into memory and labeling it as a.

We can retrieve the value from memory using the label

print a

Python programs generally consist of many statements that store and examine variables. And these variables are stored in tables. So, we can think of the basic memory model for Python as manipulation of tables.

In these tables, Python stores the names of values of all the variables. Values are things like numbers or strings. There are four “simple” types of data that Python manages:

1. Integers (<int>): whole numbers, e.g. -23, 0, 12324, 8
2. Floating Point Numbers (<float>): numbers with decimal points in them, e.g. 1.0, 5.3, 0.1
3. Booleans (<bool>): True or False
4. Strings (<str>): strings of characters, e.g. “hello”, or ’hi’

Python has different operations that apply to different types. For example, 32 + 4 performs math an results in 36. But for strings, the operation is concatenation ’32’ + ’4’ results in ’324’. And ’32’ + 4 results in an error because it doesn’t make sense to add the number 4 to a string.
2.1 Assignment statements store data in tables

When we run Python in the interpreter, there is a “main” table (I often call this a “symbol table” because it keeps track of all of the symbols/labels). Commands manipulate the entries in this table. This table has two columns: Name and Value.

One example of this manipulation is by the assignment statement. An assignment statement associates data with a label, or variable. It does this by adding a row to the symbol table.

Then we execute

georgeWeight = 32

to store the fact that my son George weighs 32 pounds.

Note that this is not the same as “equals” in mathematics. Whenever you see this “equals” sign, you should say to yourself “this is an assignment statement”. We are assigning the value 32 to the variable georgeWeight. I always say “gets”, e.g. “georgeWeight gets 32”.

When this line is executed, Python checks the symbol table. If it contains an entry named georgeWeight, it will update its value. If it does not contain and entry named georgeWeight, it will add a row with the name georgeWeight and value 32, e.g.

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>georgeWeight</td>
<td>32</td>
</tr>
</tbody>
</table>

I will illustrate the remaining points by extending this example. This example is constructed to demonstrate that data is just data. It is your responsibility as programmer to make sure the data are updated appropriately.

Now execute

bigBoy = georgeWeight > 40

Python compares the value of georgeWeight to 40, finds that 32 is not greater than 40. I.e. the statement that $32 > 40$ is False, so result is the boolean value False. This value is assigned to the variable bigBoy and the table is updated:
Suppose George grows. We change his weight to reflect that growth:

\[
\text{georgeWeight} = 42
\]

and the value of `georgeWeight` changes in the table:

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>georgeWeight</td>
<td>42 &lt;int&gt;</td>
</tr>
<tr>
<td>bigBoy</td>
<td>False &lt;bool&gt;</td>
</tr>
</tbody>
</table>

Is George big? Well, yes, he is a big boy now. But the variable `bigBoy` is still False. The entry in the table is just data – there is nothing in the table that tells us how we got that value – their is nothing in the table that indicates one variable depends on another. It is up to us, as the intelligent humans to update it.

So we again execute

\[
\text{bigBoy} = \text{georgeWeight} > 40
\]

and the table becomes

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>georgeWeight</td>
<td>42 &lt;int&gt;</td>
</tr>
<tr>
<td>bigBoy</td>
<td>True &lt;bool&gt;</td>
</tr>
</tbody>
</table>

## 2.2 Rules for Assignment

Here is a more formal description of what Python does when it executes an assignment statement.

1. Evaluate the right-hand-side – this means execute any code on the right-hand side of the equals sign. The result will be the value entered into the table.

2. Update the table

   (a) If there is already an entry for the variable on the left-hand side, then update the value for that row.
   (b) Otherwise, add a row to the table.
2.3 Modules and Functions

Numbers, strings, and booleans are not the only things stored in the Python memory model. Values of more complex types are stored as well. For example, modules and functions are also stored in symbol tables.

A *module* is a named collection of functions. For example, there is a module named turtle which provides the turtle graphics commands you will be using in lab today.

Suppose we are running Python in the interpreter mode. To have access to the module, we must import it. e.g. `import turtle`. This command adds a row to the main symbol table whose name is turtle and whose value is an arrow pointing to a module.

Sidenote: We draw “simple” or “basic” types as entries directly in the value field for an entry. More complicated types, like modules and functions live outside the table, and the entry is actually an arrow to the value.

The module type contains a symbol table, which contains entries for each of the functions defined in that module. See Figure 1 for a depiction.

To access the turtle functions, we must use what is called “dot” notation. I.e. to execute the forward method, we write `turtle.forward()`. I like the think of the dots as providing the links in a chain from table to table.

On the other hand, if we did what you did in lab last week, i.e.

```
from turtle import *
```

then all of the turtle functions are placed directly into the main symbol table. This makes the table cluttered. This isn’t a good idea, because it restricts what names you can use for variables. There is nothing to prevent you from executing a line like:

```
left = 'left'
```

which will replace the turtle function with a string. Ick!
Figure 1: Main symbol table after the turtle module has been imported. The module has its own symbol table, containing entries for each of the turtle functions/commands.