1 Administrative Topics

- Did everyone successfully submit their projects?

2 Rules for Assignment

On Friday, we talked about assignment statements in Python and how values like numbers, strings, and booleans are stored in symbol tables. For example

\[ a = 6 \]

updates a symbol table with a row with the name \( a \) and the value 6 of type \text{int}. We call \( a \) a \textit{variable}.

Let us formalize our understanding of how an assignment statement is handled by Python;

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.
3 Math Operators

There are quite a few mathematical operators built in to Python. Table 1 contains a list of binary and unary operators

<table>
<thead>
<tr>
<th>Operation</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>x + y</td>
<td>sum of x and y</td>
</tr>
<tr>
<td>x - y</td>
<td>difference of x and y</td>
</tr>
<tr>
<td>x * y</td>
<td>product of x and y</td>
</tr>
<tr>
<td>x / y</td>
<td>quotient of x and y</td>
</tr>
<tr>
<td>x % y</td>
<td>remainder of x/y</td>
</tr>
<tr>
<td>-x</td>
<td>x negated</td>
</tr>
<tr>
<td>+x</td>
<td>x unchanged</td>
</tr>
<tr>
<td>x ** y</td>
<td>x to the power y</td>
</tr>
</tbody>
</table>

Here are some interesting points regarding math in Python:

- The same precedence rules apply to the operations as in math. For operators with the same precedence, Python executes them from left to right.
- For floats, +,-,*,/ do what you expect.
- For integers +,-,* are also straightforward.
- What about / and % for integers? / results in the quotient and % results in the remainder, e.g. 5/3 is 1 and 5%3/ is 2.
- We can group together operations using parentheses the same way we can with mathematical expressions.

The rules of precedence for the binary operators are:

- ** (highest)
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 ?? 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.

```python
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:

```python
left = 'left'
```
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.

which will replace the turtle function with a string. Ick!

3.2 Math Module

In section A on Mon, we went over this section.

There are lots of mathematical functions available in the Math module. For example, sin, cos, and other trigonometric functions. Also, there are a few “rounding” functions – floor, ceil, trunc. Finally, there are some interesting one to test whether or not a variable contains an odd value, like infinity or NaN (special symbol for “Not a Number”).

Also,
import math

adds an entry in the symbol table with the name “math” and the value is of type <module>. The module is a collection of math functions. So we execute them like this:

a = math.sin(0.0)

In other words, we provide Python the directions to the function – look up “math” in the main table, then “sin” within the module.

Using this method of importing, it is easy to keep your code organized. But it does require extra writing (i.e. the “math.” before each function). There is another method of importing:

from math import *

This tells Python to find a “math” module, open it, and dump all of its contents into the current symbol table. So, instead of having one entry called “math”, we have one entry for each of the functions in the math module. This makes calling math functions more convenient, but makes organization a little tougher because everything is in one table.

Later, you will be writing code that draws things. You may want a variable named “tan” to store info about the color tan. However, the symbol “tan” is already used for the tangent function in math. So, you have two uses for the same symbol - different meanings in different contexts (a color or a math function). But you can have only one value per symbol. The easiest way to resolve that problem is to use the first method of importing. Then, the math function will be called math.tan and the color variable will be called tan.

4 Organizing Code

A Python program written for this class should be organized as follows:

1. Header comments (name of file, name of author)

2. Any import statements

3. Any function definitions (each of which should have a comment indicating what it does)
4. A comment indicating the “main” section of code is about to appear

5. The main code that you want this program to run

For example, a simple Python program that uses the turtle to draw a few squares is this:

```python
# squares.py
# Stephanie Taylor
import turtle

# Draw a square with side of the given length
def drawSquare(length):
    turtle.color((0, 0.5, 0))
    turtle.forward(length)
    turtle.left(90)
    turtle.forward(length)
    turtle.left(90)
    turtle.forward(length)
    turtle.left(90)
    turtle.forward(length)
    turtle.left(90)

# main
turtle.reset()
turtle.tracer(False)
len = input('what length? ')
drawSquare(len)
turtle.right(30)
drawSquare(100)
turtle.right(30)
drawSquare(100)
turtle.update()
raw_input("press enter when ready ")
```
5 Writing Our Own Functions

6 Writing Our Own Functions that return values

We have written functions to draw scenes with the turtle, but we haven’t needed to print or store any results from those functions. When we call the mathematical functions, there is always a value returned to us. So, how do we write functions (like math functions) that return results? The answer is that we use a “return” statement.

As a review, here is what we know about writing functions: Functions have input, instructions, and output

- We begin by writing the “header” comments. Name the function, briefly describe what it does, then give detailed information about what it expects as input (including the types), then list what it returns.
- Define the function. using the keyword “def”, the name of the function, and the parameters it takes as input
- Perform the task of the function (e.g. the “do the math”)
- Insert a return statement to return the value to the caller.

Let’s write a function that will sum two numbers and return the result:

```python
# summing.py
# Stephanie Taylor

# Function doASum - adds a and b and returns the sum
def doASum(a, b):
    result = a + b
    return result

# main
C = doASum(2, 5)
```
7 Executing Functions

I type `python cubing.py` on the Terminal command line.

Python executes the file like this

1. Reads in the definition of `doASum` and places it in the main symbol table

   main
   \[
   \begin{array}{|c|c|}
   \hline
   \text{Name} & \text{Value} \\
   \hline
   \text{doASum} & \langle \text{fcn} \rangle \\
   \hline
   \end{array}
   \]

2. Executes the code after the `#main` comment

   I like to write the line of code currently being executed directly underneath the appropriate symbol table, i.e.

   main
   \[
   \begin{array}{|c|c|}
   \hline
   \text{Name} & \text{Value} \\
   \hline
   \text{doASum} & \langle \text{fcn} \rangle \\
   \hline
   \end{array}
   \]
   \[
   c = \text{doASum}(2,5)
   \]

   Now, let’s evaluate the statement. Do you remember how assignment works? First, the right hand side is evaluated. In our case the function `cube` is called. It is given as input the values 2 and 5. The input is “passed in” to the function. A new symbol table is created called `cube`. Variables named `a` and `b` are created and assigned the values given in the argument. The new lines are added to the `doASum` symbol table.

   main
   \[
   \begin{array}{|c|c|}
   \hline
   \text{Name} & \text{Value} \\
   \hline
   \text{doASum} & \langle \text{fcn} \rangle \\
   \hline
   \end{array}
   \]
   \[
   \begin{array}{|c|c|}
   \hline
   \text{Name} & \text{Value} \\
   \hline
   a & 2<\text{int}> \\
   b & 5<\text{int}> \\
   \hline
   \end{array}
   \]
   \[
   c = \text{doASum}(2,5)
   \]

   Next, the line `result = a+b` is executed. Since this line is part of the `doASum` function, we write it underneath the symbol table for the `doASum` function:
This is an assignment statement. How do we execute assignment statements?

Evaluate the r.h.s.

Look for the symbol in the current table. If it isn’t there, create a row for it.

Put the value from the r.h.s. into the value column.

So, we add a row for `result` with the value 7. Then, the assignment statement has finished execution:

```
c = doASum(2,5)
```

Now the function gets ready to return. It returns the value in `result`.

First, we add the statement `return` result underneath the cube table:

```
c = doASum(2,5)
```
Now, we need to indicate that the value is being thrown back to the caller (i.e. the code in main). Let’s represent that by going back to the code under the main symbol table and replacing it with the return value:

Now, soASum has finished executing. We do not need its memory any more. So, we erase its symbol table:

Now, we continue evaluating our original assignment statement. We have finished evaluating the right hand side of the assignment statement $c = doASum(2,5)$. We finish executing the assignment statement by updating the main symbol table:
Phew! We are done.

Let’s reiterate what happens when a function, like cube, is executed:

1. A symbol table is created for the function
2. One row is added to the symbol table for each parameter. The values are taken from the caller.
3. The code in the function is executed.
4. Any return value is sent back to the caller.
5. The function exits and the symbol table is deleted.
6. “Control” is returned to the caller.