CS61A Lab 2: Control Flow

Week 2, 2011

Exercise 1: Assignment

Predict what Python will print in response to each of these expressions. Then try it and make sure your answer was correct, or if not, that you understand why!

```
a = 1
b = a + 1
a + b + a * b
_____________

a == b
_____________

z, y = 1, 2
print(z)
_____________

def square(x):
    x * x         # Hit enter twice
a = square(b)
print(a)
_____________

def square(y):
    return y * y # Hit enter twice
a = square(b)
print(a)
______________
```

Exercise 2: Control Flow

Predict what Python will print in response to each of these expressions. Then try it and make sure your answer was correct, or if not, that you understand why!

boolean operators

```
a > b and a == 0
```

```
_______________

a > b or a == 0

_______________

not a == 0

_______________
```

if statements

```
if a==b:
    a
else:
    b

_______________

if a == 4:
    6
elif b >= 4:
    6 + 7 + a
else:
    25

________________

if b != a: a; b  # ; lets you save a newline

_________________
```

while loops

```
n = 4
def countDown(n):
    while n > 0:
        print(n)
        n = n - 1
countDown(5)

__________________
```

Instructions:

For the next three exercises, you will be using emacs. Create a file, `lab2.py` and put the following three function definitions in this file. We will have you print this file at the end.

Exercise 3: `max()`

Define a procedure, `max(a, b)` that takes in two numbers as arguments and returns the maximum of those two numbers without using the builtin `max` function.

Exercise 4: `expt()`

Write a procedure `expt(base, power)` which implements the exponent function. For example, calling `expt(3, 2)` should return 9, and calling `expt(2, 3)` should return 8.

Exercise 5: `factor()`

Before we write our next function, let's look at the idea of integer division versus normal division.

```
Normal Division                         Integer Division

# a / b → returns a float!              # a // b → rounds down to the nearest integer
>>> 1/4                                 >>> 1//4
0.25                                    0
>>> 4/2                                 >>> 4//2
2.0                                     2
>>> 5/3                                 >>> 5//3
1.666666666667                          1
```

Thus, if we have a function "%" that gives us the remainder of dividing two numbers, we can see that the following rule applies:

```
b * (a // b) + (a % b) = a
```

Now, define a procedure `factors(n)` which takes in a number, n, and prints out all of the numbers that divide n evenly. For example, a call with n=20 should result as follows (order doesn’t matter):

```
>>> factors(20)
20
10
5
4
```

```
2
1
```

Helpful Tip: You can use the % to find if something divides evenly into a number. % gives you a remainder, as follows:

```
>>> 10 % 5
0
>>> 10 % 4
2
>>> 10 % 7
3
>>> 10 % 2
0
```

Printing:

Use the lpr command to print your `lab2.py` file. To do this, type the following command into your main xterm window (not Emacs or Python):

```
   lpr lab2.py
```

This will print the lab2 file in the room opposite from 273 Soda. You could print other files by issuing a similar command. You have 200 free pages under your account.

Exercise 6: Additional ucb.py features

At the end of last lab, we introduced the ucb.py file which the staff have provided and contains several helpful features in writing code. We started with main which helps with testing.

In order to use any features of the ucb.py file, you need to import it into your Python files by including an import statement at the top of each file:

```
        from ucb import main, trace, interact
```

Section I: A Recap of `main`

An entry point of a program is the place where execution starts happening. It's usually very convenient to be able to demarcate an entry point in a Python file for testing purposes. Say we have the following file cube.py:

```
def cube(x):
    return x * x * x

print("Should be 1:", cube(1))
print("Should be 8:", cube(2))
print("Should be 27:", cube(3))

star [123] ~ # python cube.py
Should be 1: 1
Should be 8: 8
Should be 27: 27
```

One problem with this file is that the tests aren't cleanly arranged. Plus, if I'm using Python in Emacs, then the test output will print out every time I re-load the `cube.py` file, which we might not want.

> Note: The above interaction shows how to run Python scripts from the command line. When you do `python cube.py`, Python will effectively evaluate each line of the file, which is why the print outputs show when we run `cube.py`.

Instead, it'd be much better if we had a test function that performed these tests:

```
def cube(x):
    return x * x * x

def run_tests():
    print("Should be 1:", cube(1))
    print("Should be 8:", cube(2))
    print("Should be 27:", cube(3))
```

However, now, if I run the file, nothing happens:

```
star [123] ~ # python cube.py
star [124] ~ #
```

This is because, to Python, all we've done is define two functions: `cube`, and `run_tests`. We want Python to actually do something when we do '`python cube.py`', however. So, we can specify an entry point with the `@main` annotation:

```
from ucb import main, trace, interact

def cube(x):
    return x * x * x

def run_tests():
    print("Should be 1:", cube(1))
    print("Should be 8:", cube(2))
    print("Should be 27:", cube(3))

@main
def main():
    print("Starting.")
    run_tests()
    print("Ending.")
```

```
star [123] ~ # python cube.py
Starting.
Should be 1: 1
Should be 8: 8
Should be 27: 27
Ending.
```

Section II: The `trace` feature

Let's face it - people make mistakes, especially when writing code. When trying to track down a mistake, it's usually useful to be able to see what a particular function does every time it's called. This is called trace-ing a function.

Say we have the following (incorrect) code to compute the Euclidean distance between two points:

```
def square(x):
   return 2 * x

def distance(x1,x2,y1,y2):
   return sqrt(square(x1-x2) + square(y1-y2))
```

We call the distance function with the points (3,0) and (0,0), expecting to get 3.0 back, but instead we get:

```
>>> distance(3,0,0,0)
2.449489742783178
```

Hm, that's not right! We take another look at the `distance` function, and everything seems to be structured correctly. So, the problem could be with one of the helper functions called by `distance` - we'll first examine `square`. We can trace `square` to see how `square` behaves when called by adding an '`@trace`' above the `square` definition: (Remember to import the ucb features)

```
from ucb import main, trace, interact
from math import sqrt

@trace
def square(x):
    return 2 * x

def distance(x1,x2,y1,y2):
    return sqrt(square(x1-x2) + square(y1-y2))
```

Then, when we call `distance`, we get:

```
>>> distance(3,0,0,0)
square(3):
square(3) -> 6
square(0):
square(0) -> 0
2.449489742783178
```

The `trace` output tells us when a trace'd function is called (i.e. '`square(3):`' means that the `square` function was called with argument `3`), and it also tells us what a trace'd function returns (i.e. '`square(3) -> 6`' means that `square(3)` returned `6`).

Section III: The `interact` feature

`interact` is a utility function in `ucb.py` that, when called, stops execution of the program, and opens an interactive prompt at that point in the program. This is very useful for debugging, because at this prompt, you can examine variable values during program execution.

For instance, copy the following into a new file `dist_interact.py`

```
from ucb import main, trace, interact

def distance(x1,y1,x2,y2):
    a = (x2-x1)**2
    b = (y2-y1)**2
    c = a+b
    print("== Before interact ==")
    interact()
    print("== After interact ==")
    return sqrt(c)

@main
def run():
    print("Starting.")
    print(distance(1, 1, 3, 1))
    print("Done!")
```

Run the program (either in Emacs, or in the terminal by doing '`python dist_interact.py`'). You'll notice that the '>>>' Python prompt will appear. We are currently in the middle of evaluating the body of the `distance` function - in fact, we've stopped exactly where the `interact()` function is called. Print out the values of the local variables (such as x1, y2, a, b, c) to see their values. To resume execution, do C-d (if you're on a Windows machine, you instead need to do C-z Return).

`interact()` is useful for debugging, because you get to halt execution and examine variables to see what went awry.

Fin.