Week 2 - Loops, Conditions & Functions

Tools We Can Use: Loops

Sometimes, we need to execute code multiple times, or need to run that code until some condition is met.

- `for` loops allow us to iterate through code a fixed number of times
- `while` loops allow us to repeat code until a predetermined condition is met

For Loops

The word `for` is reserved in Python to denote the start of a for loop.

```
for i in [1,2,3,4,5,6,7,8,9,10]: # repeat code below for
    print(i)                     # each value in the list
```

This code will run one time for each value in the list, and each iteration will assign the next value in the list to the variable `i`.

- We then are told to print `i`, so that each number between 1 and 10 is printed in order.

Exercise

Write a `for` loop that will square each number between 1 and 10, and print the value.

Exercise

Write a `for` loop that will square each number between 1 and 10, and print the value.

```
for i in range(1,11): # range is an iterable, with a lower
    print(i*i)        # bound included, and upper bound
                      # excluded
```

Exercise

Write **nested** `for` loops (a `for` loop inside of a `for` loop) that will square each number in a matrix stored as a list of lists. Use the following matrix:

```
myMatrix=[[1,2,3],
          [4,5,6],
          [7,8,9]]
```

Exercise

Write **nested** `for` loops (a `for` loop inside of a `for` loop) that will square each number in a matrix stored as a list of lists. Use the following matrix:

```
myMatrix=[[1,2,3],
          [4,5,6],
          [7,8,9]]
```

```
for i in range(3): # 3 is the number of elements in a row
    for j in range(3): # 3 elements per column
        myMatrix[i][j]*=myMatrix[i][j] # row THEN column
```

Note: `*=` means to multiply and then set equal

Some Shorthand

- `+=` means to add the second value, and assign the sum to the original variable
- `-=` means to subtract the second value, and assign the difference to the original variable
- `*=` means to multiply the values, and assign the product to the original variable
- `*=` means to divide the first value by the second, and assign the ratio to the original variable

While Loops

`while` loops require a logical statement.

- **While** the statement is true, the loop continues to execute
  - When the condition no longer holds, the loop terminates
- This is typically where programmers create "infinite loops"
  - Always remember to update your condition variable in each iteration!

While Loops

```
x=1
y=11
while (x<y):
    print(x)
    x+=1
```

This will print `x` every time the loop executes until `x` is no longer less than `y` (will NOT print when the two are equal).

Is the line `x+=1` necessary? Discuss with your neighbor.

Tools We Can Use: Conditions

We often want to use logical statements (a test of whether or not some condition holds) to determine what our program should do.

Here is some pseudocode (a framework of what we want our code to do, but not written in code yet):

```
if condition1 is true
    then do x
if condition2 is true instead
    then do y
otherwise
    do z
```

If, Else Statements

In programming, conditions are expressed through the keywords `if` and `else` (and also `elif` in Python)

Let's write code based on the relationship between `a` and `b`.

```
if a<b:
    print("Smaller!")
elif a>b:
    print("Larger!")
else:
    print("Equal!")
```

Set `a=12` and `b=100`. What happens? What if `b=12`?

If, Else Statements

- `if`, `else` statements are order dependent!
- The logic must be clear
  - Can't compare strings to integers, for example
- You must enumerate all possible outcomes. If not, your code might surprise you!
  - Remember, computers are stupid, and only do what they are told
  - This is the "Garbage in, garbage out" principle

From Last Week

```
import numpy as np

def manhattanDistance(coord1, coord2):
        dist = 0
        errorstring = "Coordinate dimension mismatch."
        if len(coord1)==len(coord2):
                for i in range(len(coord1)):
                        dist+=np.abs(coord1[i]-coord2[i])
                return dist
        else:
                raise runtimeError(errorstring)
```

From Last Week

```python
def manhattanDistance(coord1, coord2):
    ...
    return dist
```

Let's focus on one part of that code snippet

- Defines a function called `manhattanDistance`, as well as its arguments
- Functions are extremely powerful!

Why Use Functions?

In Python (and most languages), we can just write code as a sequence of commands, and everything will be fine.

```
import numpy as np

myCoord1 = [10,20,30]
myCoord2 = [1,2,3]
dist = 0

dist+=np.abs(myCoord1[0]-myCoord2[0])
dist+=np.abs(myCoord1[1]-myCoord2[1])
dist+=np.abs(myCoord1[2]-myCoord2[2])
```

Why Use Functions?

What might go wrong with the code on the previous slide?

- What if I want to use a new set of coordinates with 4 dimensions?
- What if I don't notice that my coordinates do not have the same number of dimensions?
- What if I want to run that code as part of antother program in a different file?
- What if I want to use Euclidean Distance in the future?

Nature of Functions

- Allow for reuse of code
  - Can import this code in other programs!
- Help us to organize our code
- Are limited in **scope** (more on that soon!)
- Allow us to quickly make broad changes

Starting to Write Functions

```
def myFunction(arguments_go_here):
```

First, we need to use the `def` statement to declare our function.

Later, after we have completed the code that runs inside of the function, we write our return statement:

```
        return objects_to_be_returned
```

Exercise

Write a function that returns the product of two numbers (note: the product of x and y is $x \times y$). Name the function `product`

- What is the result of `product(2,5)`?
- What is the result of `product(2.71828,5)`?
- What is the result of `product("Howdy!",3)`?
  - How about `product(3,"Howdy!")`?

Exercise, Part 2!

Write a function that utilizes YOUR `product` function to calculate the area of a circle with radius `r` (note: area is calculated as πr^2). Call that function `areaCircle`

- What is the result of `areaCircle(2)`?
- What is the result of `areaCircle(2.71828)`?
- What is the result of `areaCircle("Howdy!")`?

Observations

When we use the function `product`, we are able to use a string as one argument. Why?

- Python is able to determine that the multiplier function `string * y` means that we want to repeat a string y times.
- We need to use this carefully, however, as we discover when we try to determine the area of a circle with radius `"Howdy!"`. That really doesn't make sense, and Python agrees, giving us an error.

Observations

The function `areaCircle` can be created by utilizing our `product` function:

```
def areaCircle(r):
    r2 = product(r,r)
    return product(r2, 3.1415)
```

or, even more succinctly,

```
def areaCircle(r):
    return product(product(r,r), 3.1415)
```

Observations

- We can use functions inside of functions
- Use small functions to build part of a whole
- We can even use functions **recursively**

Recursive Functions

Try writing a function to calculate Fibonacci numbers.

$$F_0 = 0$$

$$F_1 = F_2 = 1$$

$$F_n = F_{n-1} + F_{n-2}$$

How can we write a function to determine an arbitrary Fibonacci number?

Recursive Functions

```python
def fibonacci(n):
    if n==0:
        return 0
    elif n==1:
        return 1
    elif n==2:
        return 1
    else:
        return fibonacci(n-1) + fibonacci(n-2)
```

This function is **recursive** because it calls *itself* in order to complete its own execution.

Recursive Functions

Calling `fibonacci(5)` will trigger the following procedure:

- Because `n` is greater than 2, `fibonacci(n-1)` and `fibonacci(n-2)` are called
    - This would be `fibonacci(4)` and `fibonacci(3)`
- `fibonacci(4)` calls `fibonacci(3)` and `fibonacci(2)`
    - `fibonacci(2)` returns a value of `1`, and `fibonacci(3)` calls `fibonacci(2)` and `fibonacci(1)`, which both return values of `1`, causing `fibonacci(3)` to return a value of `2`
      ...

Recursive Functions

- The returned values of `fibonacci(3)` (`2`) and `fibonacci(2)` (`1`) lead `fibonacci(4)` to return a value of 3
- Remember that `fibonacci(3)` was also called by `fibonacci(5)`, and again will return a value of `2`
- `fibonacci(5)` thus returns the sum of 3 and 2 (the results of `fibonacci(4)` and `fibonacci(3)`, respectively)

Calling `fibonacci(5)` utilized the `fibonacci(n)` function 9 times, and did so **recursively**

Summary of Today

- We can use loops, conditional statements, and function definitions to quickly create powerful and complex code
- Using these tools requires us to **be careful**, since computers will follow instructions LITERALLY, and do not adapt to our errors
    - Otherwise we can get infinite loops (call `fibonacci(n)` instead of `fibonacci(n-1)` in our `else` statement)

For Lab Today

Write a function `manyOptions` that takes x and y as input. Be sure to write all operations on your own (don't use other libraries!).

- If `x` equals 1, return the factorial of `y`
- If `x` equals 2, print the sum of squares of all integers between 1 and `y`
- If `x` equals 3, print the first `y` values of Hofstadter's Q-Sequence, starting with 1.
- If `x` has any other value, print a statement saying that the value of `x` is invalid