Week 2 Coding Lecture 2: Conditional statements

When you are programming, you will often encounter situations where you want your code to do one thing in some contexts and another thing in other contexts. We already saw one example in the last lecture: We wanted to be able to stop our loop if `fib(n)` was too large. For a simpler example, you might want to calculate the absolute value of a number. The absolute value is defined as

$$|x| = \begin{cases} x & \text{if } x \geq 0, \\ -x & \text{if } x < 0. \end{cases}$$

In English, this means that the absolute value of x is the same value if x is not negative, but you have to flip the sign if x is negative.

This general structure is so common that python (along with essentially every other programming language) has a special syntax to handle it. In python, this is called an `if` statement. In its simplest form, an `if` statement looks like this:

`if condition:

```
 # Code to be run if the condition is true`
```

The word `if` is not a variable - it is a reserved word that tells python that this is an `if` statement. The word `condition` can be a variable, or it can be replaced with a more complicated line of code. For the moment, we will assume that the code in `condition` evaluates to a `bool` variable.

We have not encountered the type `bool` before, but it is very simple. A variable of type `bool` can only have two different values, `True` or `False`. These are also keywords in python, so you can create your own `bool` variables. For example,

```
x = True
print(x)
```

```
    True
```

```
print(type(x))
```

```
print(type(x))
```

```
<class 'bool'>
```

```
y = False
print(y)
```

```
False
```

```
print(type(y))
```

```
<class 'bool'>
```

When python encounters an `if` satement, it evaluates the code in `condition` and then does one of two things. If `condition` is True, then python runs the indented code after the `if` statement, and then continues with any other un-indented code below. If `condition` is False, then python skips the indented code and goes directly to the un-indented code after the block.

Here are two prototypical examples:

```
x = True
print("The code before the if block is always run.")
if x:
    print("The code in the if block is run because x is True.")

print("The code after the if block is always run.")
```

```
The code before the if block is always run.
The code in the if block is run because x is True.
The code after the if block is always run.
```

```
y = False
print("The code before the if block is always run.")
if y:
    print("The code in the if block is not run because y is False.")

print("The code after the if block is always run.")
```

```
The code before the if block is always run.
The code after the if block is always run.
```

To make use of this syntax, we have to know how to write code that produces `bool` variables. Python has a handful of useful operators for this task. You are probably already very familiar with all of these.

Equality

The `==` operator checks if two values are equal. If they are equal, the entire expression evaluates to True. If they are not equal, the expression evaluates to False. For example,

```
x = 3
y = 3
z = 2

print(x == y)
```

```
True
```

```
print(x == z)
```

```
False
```

Notice that this is a double equals sign, not a single equals sign. The single equals sign is already used for assignment. It is very common to accidentally use assignment when you meant to use equality. For example, you might write

```
if (x = z):
    print("x is equal to z")
```

```
  File "<ipython-input-61-cb5767e81f92>", line 1
    if (x = z):
          ^
```

As you can see, python produces an error when this happens.

SEARCH STACK OVERFLOW

Greater

The > operator checks if the left value is greater than the right value. If the left value is greater, the expression evaluates to True. Otherwise, it evaluates to False.

[ ] ↳ *3 cells hidden*

Less

The < operator checks if the left value is less than the right value. If the left value is less, then the expression evaluates to True. Otherwise, it evaluates to False.

[ ] ↳ *3 cells hidden*

Greater-equal

The >= operator checks if the left value is greater than or equal to the right value. If the left value is greater or equal, the condition evaluates to True. Otherwise, it evaluates to False.

[ ] ↳ *3 cells hidden*

Less-equal

The <= operator checks if the left value is less or equal to the right value. If the left value is less or equal, the condition evaluates to True. Otehrwise, it evaluates to False.

[ ] ↳ *3 cells hidden*

Absolute value

We are now in a position to calculate absolute value in code. We can write

```
x = 10
if x >= 0:
    absolute_value_of_x = x
if x < 0:
    absolute_value_of_x = -x
print("The absolute value of x is: ")
print(absolute_value_of_x)
```

```
The absolute value of x is:
10
```

Likewise,

```
x = -5
if x >= 0:
    absolute_value_of_x = x
if x < 0:
    absolute_value_of_x = -x
print("The absolute value of x is: ")
print(absolute_value_of_x)
```

```
The absolute value of x is:
5
```

Else

Notice that we had two mutually exclusive conditions in this code (either x is non-negative or it is negative, but not both) and we wanted to run different code in either case. Our solution is ok, but this comes up so often that python has a shortcut: The `else` statement. In general, `else` statements work like this:

`if condition:

```
# Code to be run if the condition is true
```

else:

```
# Code to be run if the condition is false`
```

When python encounters a block like this, it first evaluates `condition`. If `condition` is True, then python executes the code in the first indented block, then skips the second indented block and goes directly to un-indented code after the `if` and `else` blocks. If `condition` is False, then python skips the code in the first indented block and goes directly to the second indented block, then continues with un-indented code after both blocks.

We can therefore clean up our code like this:

[ ] ↳ *2 cells hidden*

Elif

There is actually one more level of generality for `if` statements. If we have more than two cases, but we only ever want to run one block of code, we can use the keyword `elif`. The general case looks like this:

`if condition_0:

```
# code block 0
```

elif condition_1:

```
# code block 1
```

... elif condition_n:

```
 # code block n
```

else:

```
 # code block n+1`
```

When python encounters a block like this, it checks each of the conditions in turn. The first time it encounters a condition that evaluates to True, python executes the corresponding indented block of code and then skips directly to un-indented code after the block. If all of the conditions evaluate to False, then python executes the last block of code (after the `else`) and then goes on to any un-indented code below. You can use as many `elif` statements as you want after an `if`, but only one `else`. For example:

[ ] ↳ *1 cell hidden*

And, Or and Not

There are two very common ways to combine conditions: The `and` operator and the `or` operator. The `and` operator evaluates to True if both the left and right value are True and evaluates to False otherwise. For instance,

```
print(True and True)
```

```
    True
```

```
print(True and False)
```

```
    False
```

```
print(False and True)
```

```
    False
```

```
print(False and False)
```

```
print(False and False)
```

```
False
```

The `or` operator evaluates to True if the left or right value (or both) are True and it evaluates to False otherwise. For instance,

```
print(True or True)
```

```
True
```

```
print(True or False)
```

```
True
```

```
print(False or True)
```

```
True
```

```
print(False or False)
```

```
False
```

The `not` operator flips the value of a `bool`. That is, `not True` is `False` and `not False` is `True`.

```
print(not True)
```

```
False
```

```
print(not False)
```

```
True
```

Non-bool conditions

There is actually no requirement that the conditions in an `if` statement evaluate to `bool` variables. The exact rules for what happens when a condition has some other type are up to whoever wrote the code for that type, but there is a very common rule (which will apply to all the types we care about).

If `condition` evaluates to zero, then the `if` statement will treat it as if it were `False`. If `condition` evaluates to any number other than zero, then the `if` statement will treat it as if it were `True`.

For example,

```
x = 10
if x:
    print("x is not zero, so this gets printed")
```

```
 x is not zero, so this gets printed
```

```
y = 0
if y:
    print("y is zero, so this doesn't get printed")
```

Fibonacci numbers

We are now in a position to solve the last problem from the previous lecture. Recall that we were trying to calculate all of the Fibonacci numbers less than 1,000,000. We already have code to calculate the first N Fibonacci numbers, where N is a fixed number:

```
import numpy as np

N = 20
fib = np.zeros(N)
fib[0] = 1
fib[1] = 1

for n in range(2, N):
    fib[n] = fib[n - 1] + fib[n - 2]
```

```
print(fib)
```

```
[1.000e+00 1.000e+00 2.000e+00 3.000e+00 5.000e+00 8.000e+00 1.300e+01
 2.100e+01 3.400e+01 5.500e+01 8.900e+01 1.440e+02 2.330e+02 3.770e+02
 6.100e+02 9.870e+02 1.597e+03 2.584e+03 4.181e+03 6.765e+03]
```

We now want to adjust this code so that it stops once we find a number larger than 1,000,000. To solve this, we really just have to add a statement to our loop saying "if fib[n] is at least 1,000,000 then stop the loop". Translating this into an `if` statement is pretty straightforward. The only other thing we need to know is that the keyword `break` tells python to stop whatever loop it is running. We can now write

```
import numpy as np

N = 20
fib = np.zeros(N)
fib[0] = 1
fib[1] = 1

for n in range(2, N):
    fib[n] = fib[n - 1] + fib[n - 2]
    if fib[n] >= 1000000:
        break

print(fib)
```

```
[1.000e+00 1.000e+00 2.000e+00 3.000e+00 5.000e+00 8.000e+00 1.300e+01
 2.100e+01 3.400e+01 5.500e+01 8.900e+01 1.440e+02 2.330e+02 3.770e+02
 6.100e+02 9.870e+02 1.597e+03 2.584e+03 4.181e+03 6.765e+03]
```

This code is almost done, but there are a couple of small issues. Most obviously, we do not have the right answer. The problem is that there are two different ways for our loop to stop. One reason is that `fib[n]` becomes bigger than 1,000,000. The other is that n gets to 20 and the `for` loop finishes as usual. In our case, we just went through all 20 steps of the `for` loop without ever finding a large enough Fibonacci number. We can mostly solve this issue by making N larger. For example, we already saw in the last lecture that the 200th Fibonacci number is larger than 1,000,000, so if we set `N = 200` then we know that we will calculate enough values.

```
import numpy as np

N = 200
fib = np.zeros(N)
fib[0] = 1
fib[1] = 1

for n in range(2, N):
    fib[n] = fib[n - 1] + fib[n - 2]
    if fib[n] >= 1000000:
        break

print(fib)
```

```
[1.000000e+00 1.000000e+00 2.000000e+00 3.000000e+00 5.000000e+00
 8.000000e+00 1.300000e+01 2.100000e+01 3.400000e+01 5.500000e+01
 8.900000e+01 1.440000e+02 2.330000e+02 3.770000e+02 6.100000e+02
 9.870000e+02 1.597000e+03 2.584000e+03 4.181000e+03 6.765000e+03
 1.094600e+04 1.771100e+04 2.865700e+04 4.636800e+04 7.502500e+04
 1.213930e+05 1.964180e+05 3.178110e+05 5.142290e+05 8.320400e+05
 1.346269e+06 0.000000e+00 0.000000e+00 0.000000e+00 0.000000e+00
 0.000000e+00 0.000000e+00 0.000000e+00 0.000000e+00 0.000000e+00
 0.000000e+00 0.000000e+00 0.000000e+00 0.000000e+00 0.000000e+00
 0.000000e+00 0.000000e+00 0.000000e+00 0.000000e+00 0.000000e+00
 0.000000e+00 0.000000e+00 0.000000e+00 0.000000e+00 0.000000e+00
 0.000000e+00 0.000000e+00 0.000000e+00 0.000000e+00 0.000000e+00
 0.000000e+00 0.000000e+00 0.000000e+00 0.000000e+00 0.000000e+00
 0.000000e+00 0.000000e+00 0.000000e+00 0.000000e+00 0.000000e+00
 0.000000e+00 0.000000e+00 0.000000e+00 0.000000e+00 0.000000e+00
 0.000000e+00 0.000000e+00 0.000000e+00 0.000000e+00 0.000000e+00
 0.000000e+00 0.000000e+00 0.000000e+00 0.000000e+00 0.000000e+00
 0.000000e+00 0.000000e+00 0.000000e+00 0.000000e+00 0.000000e+00
 0.000000e+00 0.000000e+00 0.000000e+00 0.000000e+00 0.000000e+00
 0.000000e+00 0.000000e+00 0.000000e+00 0.000000e+00 0.000000e+00
 0.000000e+00 0.000000e+00 0.000000e+00 0.000000e+00 0.000000e+00
 0.000000e+00 0.000000e+00 0.000000e+00 0.000000e+00 0.000000e+00
 0.000000e+00 0.000000e+00 0.000000e+00 0.000000e+00 0.000000e+00
 0.000000e+00 0.000000e+00 0.000000e+00 0.000000e+00 0.000000e+00
 0.000000e+00 0.000000e+00 0.000000e+00 0.000000e+00 0.000000e+00
```

```
 0.000000e+00 0.000000e+00 0.000000e+00 0.000000e+00 0.000000e+00
 0.000000e+00 0.000000e+00 0.000000e+00 0.000000e+00 0.000000e+00
 0.000000e+00 0.000000e+00 0.000000e+00 0.000000e+00 0.000000e+00
 0.000000e+00 0.000000e+00 0.000000e+00 0.000000e+00 0.000000e+00
 0.000000e+00 0.000000e+00 0.000000e+00 0.000000e+00 0.000000e+00
 0.000000e+00 0.000000e+00 0.000000e+00 0.000000e+00 0.000000e+00
 0.000000e+00 0.000000e+00 0.000000e+00 0.000000e+00 0.000000e+00
 0.000000e+00 0.000000e+00 0.000000e+00 0.000000e+00 0.000000e+00
 0.000000e+00 0.000000e+00 0.000000e+00 0.000000e+00 0.000000e+00
 0.000000e+00 0.000000e+00 0.000000e+00 0.000000e+00 0.000000e+00
 0.000000e+00 0.000000e+00 0.000000e+00 0.000000e+00 0.000000e+00
 0.000000e+00 0.000000e+00 0.000000e+00 0.000000e+00 0.000000e+00
 0.000000e+00 0.000000e+00 0.000000e+00 0.000000e+00 0.000000e+00
 0.000000e+00 0.000000e+00 0.000000e+00 0.000000e+00 0.000000e+00]
```

Of course, now we have a different issue. Now the vector has all of the Fibonacci numbers we wanted (plus one more that is over 1,000,000), but it also has more than 100 trailing zeros. This is because there are actually only 30 Fibonacci numbers less than 1,000,000, not 200. We could of course just set N to 30, but that would destroy the point of using an `if` statement in the first place. A better solution is just to cut off the extra zeros once we are done.

How do we know how many entries of our array to keep? The trick is to notice that `n` is still defined after the loop finishes, so the final value of `n` tells you how many steps were completed. In our case, it is easy to check that `n` is 30 at the end of this code block, which means that we filled in every entry up to index 30. We actually don't want the last entry we filled in, because `fib[n]` is over 1,000,000, so we only want to keep the entries of x up to index 29. We can therefore use this solution:

```
import numpy as np

N = 200
fib = np.zeros(N)
fib[0] = 1
fib[1] = 1

for n in range(2, N):
    fib[n] = fib[n - 1] + fib[n - 2]
    if fib[n] >= 1000000:
        break

fib = fib[:n-1]
```

```
print(fib)
```

```
[1.00000e+00 1.00000e+00 2.00000e+00 3.00000e+00 5.00000e+00 8.00000e+00
 1.30000e+01 2.10000e+01 3.40000e+01 5.50000e+01 8.90000e+01 1.44000e+02
 2.33000e+02 3.77000e+02 6.10000e+02 9.87000e+02 1.59700e+03 2.58400e+03
 4.18100e+03 6.76500e+03 1.09460e+04 1.77110e+04 2.86570e+04 4.63680e+04
 7.50250e+04 1.21393e+05 1.96418e+05 3.17811e+05 5.14229e+05]
```

You should take some time to study the above code. It is a very common method and we will use the same structure repeatedly in this class. The general idea is to initialize storage (in this case, the `fib` vector) for our results, then use a loop to calculate each value in turn, then use a combination of `if` and `break` to stop our loop once we have all the answers we need.

There is still one small nuisance with this code: We need to make sure that we make `N` large enough. Our solution was essentially just to guess and check until we found a sufficiently large value of `N`, but python has another coding construct for this purpose.

While loops

Up until now, we have been using `for` loops, but python actually has another type of loop called the `while` loop. The general syntax for a while loop is

`while condition:

```
 # Code to repeat`
```

When python encounters a `while` loop, it checks to see if the condition is True. If it is, then python runs the indented block of code. Python then returns to the top of the `while` loop and starts over. If the condition is ever False, then python will instead skip to the next line of un-indented code.

You should think of a `while` loop as a repeating `if` block. Python repeats the `if` block over and over until the condition becomes False.

[ ] ↳ *3 cells hidden*

Infinite loops

There is one very important point that you have to keep in mind when working with `while` loops. It is possible that the condition in your loop never becomes False. If this happens, then the loop will just repeat forever and your code will never stop running. As a silly example, this code prints the number 1 forever:

```
while True: print(1)
```

We call such an example an *infinite loop*. If you ever accidentally create one (and you almost certainly will if you use `while` loops), you can force PyCharm to stop running your code with the shortcut Ctrl+F2.