L6

June 16, 2017

1 Lecture 6: Conditionals and Exceptions

CSCI 1360E: Foundations for Informatics and Analytics

1.1 Overview and Objectives

In this lecture, we'll go over how to make "decisions" over the course of your code depending on the values certain variables take. We'll also introduce exceptions and how to handle them gracefully. By the end of the lecture, you should be able to

- Build arbitrary conditional hierarchies to test a variety of possible circumstances
- Construct elementary boolean logic statements
- Catch basic errors and present meaningful error messages in lieu of a Python crash

1.2 Part 1: Conditionals

Up until now, we've been somewhat hobbled in our coding prowess; we've lacked the tools to make different decisions depending on the values our variables take.

For example: how do you find the maximum value in a list of numbers?

```
In [1]: x = [51, 65, 56, 19, 11, 49, 81, 59, 45, 73]
```

If we want to figure out the maximum value, we'll obviously need a loop to check each element of the list (which we know how to do), and a variable to store the maximum.

```
In [2]: max_val = 0
        for element in x:

            # ... now what?

            pass
```

We also know we can check relative values, like `max_val < element`. If this evaluates to `True`, we know we've found a number in the list that's bigger than our current candidate for maximum value. But how do we execute code until this condition, and *this condition alone*?

Enter `if / elif / else` statements! (otherwise known as "conditionals")

We can use the keyword `if`, followed by a statement that evaluates to either `True` or `False`, to determine whether or not to execute the code. For a straightforward example:

```
In [3]: x = 5
        if x < 5:
            print("How did this happen?!")  # Spoiler alert: this won't happen.

        if x == 5:
            print("Working as intended.")

Working as intended.
```

In conjunction with `if`, we also have an `else` clause that we can use to execute whenever the `if` statement doesn't:

```
In [4]: x = 5
        if x < 5:
            print("How did this happen?!")  # Spoiler alert: this won't happen.
        else:
            print("Correct.")

Correct.
```

This is great! We can finally finish computing the maximum element of a list!

```
In [5]: x = [51, 65, 56, 19, 11, 49, 81, 59, 45, 73]
        max_val = 0
        for element in x:
            if max_val < element:
                max_val = element

        print("The maximum element is: {}".format(max_val))

The maximum element is: 81
```

Let's pause here and walk through that code.

`x = [51, 65, 56, 19, 11, 49, 81, 59, 45, 73]` - This code defines the list we want to look at.

`max_val = 0` - And this is a placeholder for the eventual maximum value.

`for element in x:` - A standard `for` loop header: we're iterating over the list `x`, one at a time storing its elements in the variable `element`.

`if max_val < element:` - The first line of the loop body is an `if` statement. This statement asks: is the value in our current `max_val` placeholder smaller than the element of the list stored in `element`?

`max_val = element` - If the answer to that `if` statement is `True`, then this line executes: it sets our placeholder equal to the current list element.

Let's look at slightly more complicated but utterly classic example: assigning letter grades from numerical grades.

```
In [6]: student_grades = {
            'Jen': 82,
            'Shannon': 75,
            'Natasha': 94,
            'Benjamin': 48,
        }
```

We know the 90-100 range is an "A", 80-89 is a "B", and so on. How would we build a conditional to assign letter grades?

The third and final component of conditionals is the `elif` statement (short for "else if").

`elif` allows us to evaluate as many options as we'd like, all within *the same conditional context* (this is important). So for our grading example, it might look like this:

```
In [7]: letter = ''
        for student, grade in student_grades.items():
            if grade >= 90:
                letter = "A"
            elif grade >= 80:
                letter = "B"
            elif grade >= 70:
                letter = "C"
            elif grade >= 60:
                letter = "D"
            else:
                letter = "F"

            print(student, letter)
```

```
Benjamin F
Jen B
Natasha A
Shannon C
```

Ok, that's neat. But there's still one more edge case: what happens if we want to enforce multiple conditions *simultaneously*?

To illustrate, let's go back to our example of finding the maximum value in a list, and this time, let's try to find the *second*-largest value in the list. For simplicity, let's say we've already found the largest value.

```
In [8]: x = [51, 65, 56, 19, 11, 49, 81, 59, 45, 73]
        max_val = 81  # We've already found it!
        second_largest = 0
```

Here's the rub: we now have *two* constraints to enforce--the second largest value needs to be larger than pretty much everything in the list, but also needs to be *smaller* than the maximum value. Not something we can encode using `if` / `elif` / `else`.

Instead, we'll use two more keywords integral to conditionals: `and` and `or`.

You've already seen `and`: this is used to join multiple boolean statements together in such a way that, if one of the statements is `False`, the *entire* statement is `False`.

```
In [9]: True and True and True and True and True and True and False
```

```
Out[9]: False
```

One False ruins the whole thing.

However, we haven't encountered or before. How do you think it works?

Here's are two examples:

```
In [10]: True or True or True or True or True or True or False
```

```
Out[10]: True
```

```
In [11]: False or False or False or False or False or False or True
```

```
Out[11]: True
```

Figured it out?

Whereas and needs *every* statement it joins to be True in order for the whole statement to be True, only one statement among those joined by or needs to be True for everything to be True.

How about this example?

```
In [12]: (True and False) or (True or False)
```

```
Out[12]: True
```

(Order of operations works the same way!)

Getting back to conditionals, then: we can use this *boolean logic* to enforce multiple constraints simultaneously.

```
In [13]: for element in x:
             if second_largest < element and element < max_val:
                 second_largest = element

         print("The second-largest element is: {}".format(second_largest))
```

```
The second-largest element is: 73
```

Let's step through the code.

```
for element in x:
    if second_largest < element and element < max_val:
        second_largest = element
```

- The first condition, second_largest < element, is the same as before: if our current estimate of the second largest element is smaller than the latest element we're looking at, it's definitely a candidate for second-largest.

- The second condition, element < max_val, is what ensures we don't just pick the largest value again. This enforces the constraint that the current element we're looking at is also *less than* the maximum value.

- The `and` keyword glues these two conditions together: it *requires that they BOTH* be `True` before the code inside the statement is allowed to execute.

It would be easy to replicate this with "nested" conditionals:

```
In [14]: second_largest = 0
         for element in x:
             if second_largest < element:
                 if element < max_val:
                     second_largest = element

         print("The second-largest element is: {}".format(second_largest))
```

```
The second-largest element is: 73
```

...but your code starts getting a little unwieldy with so many indentations.

You can glue as many comparisons as you want together with `and`; the whole statement will only be `True` if *every single condition* evaluates to True. This is what `and` means: *everything* must be True.

The other side of this coin is `or`. Like `and`, you can use it to glue together multiple constraints. Unlike `and`, the whole statement will evaluate to True *as long as at least ONE condition is True*. This is far less stringent than `and`, where *ALL* conditions had to be True.

```
In [15]: numbers = [1, 2, 5, 6, 7, 9, 10]
         for num in numbers:
             if num == 2 or num == 4 or num == 6 or num == 8 or num == 10:
                 print("{} is an even number.".format(num))
```

```
2 is an even number.
6 is an even number.
10 is an even number.
```

In this contrived example, I've glued together a bunch of constraints. Obviously, these constraints are mutually exclusive; a number can't be equal to both 2 and 4 at the same time, so `num == 2 and num == 4` would never evaluate to True. However, using `or`, only one of them needs to be True for the statement underneath to execute.

There's a little bit of intuition to it.

- "I want this AND this" has the implication of both at once.
- "I want this OR this" sounds more like either one would be adequate.

One other important tidbit, concerning not only conditionals, but also lists and booleans: the `not` keyword.

An often-important task in data science, when you have a list of things, is querying whether or not some new piece of information you just received is already in your list. You could certainly loop through the list, asking "is my new_item == list[item i]?". But, thankfully, there's a better way:

```
In [16]: import random
         list_of_numbers = [i for i in range(10)]  # Generates 10 random numbers, between 1 and
         if 13 not in list_of_numbers:
             print("Aw man, my lucky number isn't here!")
```

```
Aw man, my lucky number isn't here!
```

Notice a couple things here--

- List comprehensions make an appearance! Can you parse it out?
- The `if` statement asks if the number 13 is NOT found in `list_of_numbers`
- When that statement evaluates to `True`--meaning the number is NOT found--it prints the message.

If you omit the `not` keyword, then the question becomes: "is this number in the list?"

```
In [17]: import random
         list_of_numbers = [i for i in range(10)]  # Generates 10 random numbers, between 1 and
         if 13 in list_of_numbers:
             print("Somehow the number 13 is in a list generated by range(10)")
```

Nothing is printed in this case, since our conditional is asking if the number 13 was in the list. Which it's not.

Be careful with this. Typing issues can hit you full force here: if you ask:
`if 0 in some_list`
and it's a list of *floats*, then this operation will always evaluate to `False`.

Similarly, if you ask `if "shannon" in name_list`, it will look for the precise string `"shannon"` and return `False` even if the string `"Shannon"` is in the list. With great power, etc etc.

1.3 Part 2: Error Handling

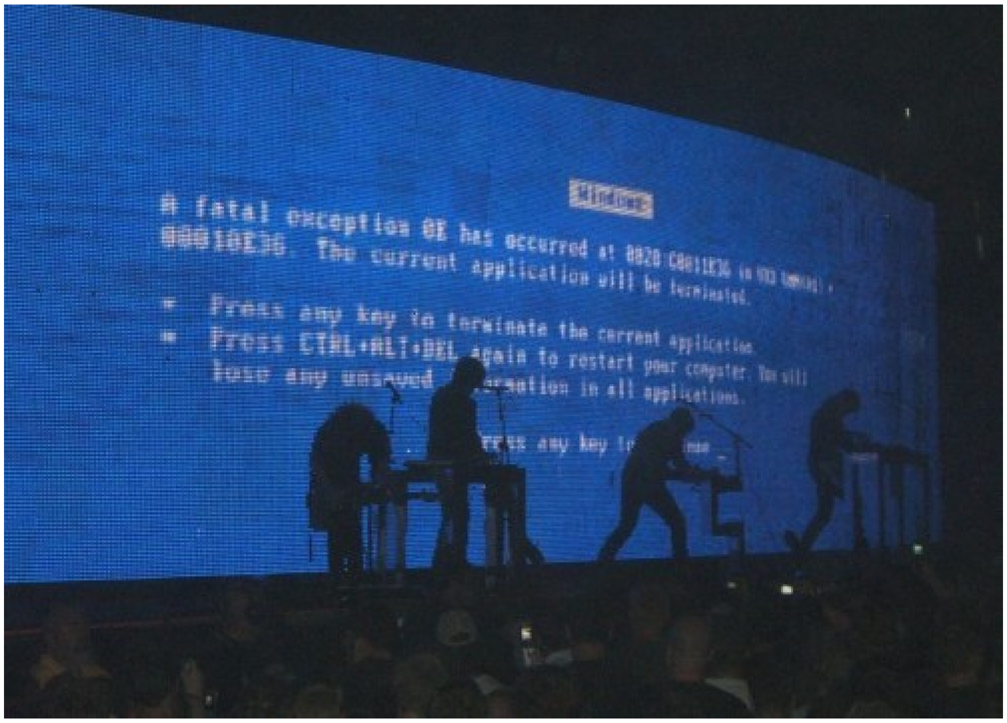
Yes, errors: plaguing us since Windows 95 (but really, since well before then).

By now, I suspect you've likely seen your fair share of Python crashes.

- `NotImplementedError` from the homework assignments
- `TypeError` from trying to multiply an integer by a string
- `KeyError` from attempting to access a dictionary key that didn't exist
- `IndexError` from referencing a list beyond its actual length

or any number of other error messages. These are the standard way in which Python (and most other programming languages) handles error messages.

The error is known as an **Exception**. Some other terminology here includes:


Windows
A fatal exception 0E has occurred at
The current application will be terminated.
Press any key to terminate the current application.
Press CTRL+ALT+DEL again to restart your computer.

bsod

- An exception is *raised* when such an error occurs. This is why you see the code snippet `raise NotImplementedError` in your homeworks. In other languages such as Java, an exception is "thrown" instead of "raised", but the meanings are equivalent.
- When you are writing code that could potentially raise an exception, you can also write code to *catch* the exception and handle it yourself. When an exception is caught, that means it is handled without crashing the program.

Here's a fairly classic example: **divide by zero!**

Let's say we're designing a simple calculator application that divides two numbers. We'll ask the user for two numbers, divide them, and return the quotient. Seems simple enough, right?

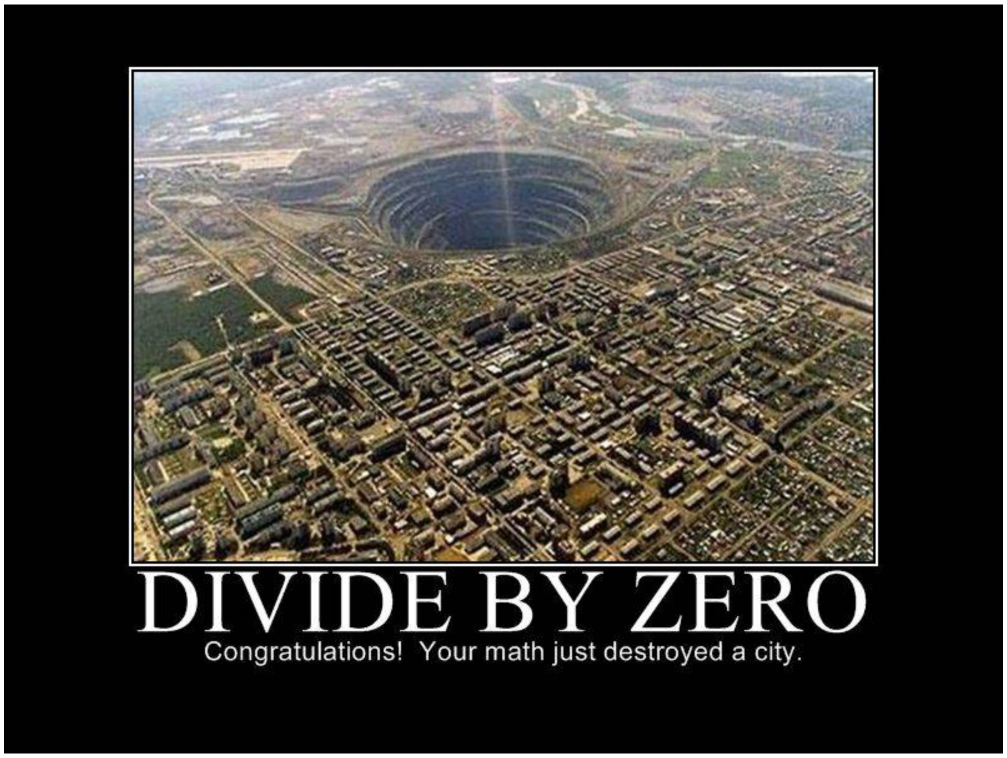
```
In [18]: def divide(x, y):
             return x / y

In [26]: divide(11, 0)
```

```
        ---------------------------------------------------------------------------

        ZeroDivisionError                         Traceback (most recent call last)

        <ipython-input-26-b8a694e48011> in <module>()
    ----> 1 divide(11, 0)
```



dbz

```
    <ipython-input-18-0f39b0c6b46e> in divide(x, y)
      1 def divide(x, y):
----> 2     return x / y


    ZeroDivisionError: division by zero
```

D'oh! The user fed us a 0 for the denominator and broke our calculator. Meanie-face.

So we know there's a possibility of the user entering a 0. This could be malicious or simply by accident. Since it's only one value that could crash our app, we could in principle have an `if` statement that checks if the denominator is 0. That would be simple and perfectly valid.

But for the sake of this lecture, let's assume we want to try and catch the `ZeroDivisionError` ourselves and handle it gracefully.

To do this, we use something called a `try / except` block, which is very similar in its structure to `if / elif / else` blocks.

First, put the code that could potentially crash your program inside a `try` statement. Under that, have a `except` statement that defines

1. A variable for the error you're catching, and
2. Any code that dictates how you want to handle the error

```
In [20]: def divide_safe(x, y):
             quotient = 0
```

```
    try:
        quotient = x / y
    except ZeroDivisionError:
        print("You tried to divide by zero. Why would you do that?!")
    return quotient
```

Now if our user tries to be snarky again--

```
In [21]: divide_safe(11, 0)
```

```
You tried to divide by zero. Why would you do that?!
```

```
Out[21]: 0
```

No error, no crash! Just a "helpful" error message.

Like conditionals, you can also create multiple `except` statements to handle multiple different possible exceptions:

```
In [22]: import random  # For generating random exceptions.
         num = random.randint(0, 1)
         try:
             # code for something can cause multiple exceptions
             pass
         except NameError:
             print("Caught a NameError!")
         except ValueError:
             print("Nope, it was actually a ValueError.")
```

Also like conditionals, you can handle multiple errors simultaneously. If, like in the previous example, your code can raise multiple exceptions, but you want to handle them all the same way, you can stack them all in one `except` statement:

```
In [23]: import random  # For generating random exceptions.
         num = random.randint(0, 1)
         try:
             if num == 1:
                 raise NameError("This happens when you use a variable you haven't defined")
             else:
                 raise ValueError("This happens when you try to multiply a string")
         except (NameError, ValueError):  # MUST have the parentheses!
             print("Caught...well, some kinda error, not sure which.")
```

```
Caught...well, some kinda error, not sure which.
```

If you're like me, and you're writing code that you know could raise one of several errors, but are too lazy to look up *specifically* what errors are possible, you can create a "catch-all" by just not specifying anything:

```
In [24]: import random  # For generating random exceptions.
         num = random.randint(0, 1)
         try:
             if num == 1:
                 raise NameError("This happens when you use a variable you haven't defined")
             else:
                 raise ValueError("This happens when you try to multiply a string")
         except:
             print("I caught something!")
```

```
I caught something!
```

Finally--and this is really getting into what's known as *control flow* (quite literally: "controlling the flow" of your program)--you can tack an `else` statement onto the very end of your exception-handling block to add some final code to the handler.

Why? This is code that is only executed if **NO** exception occurs. Let's go back to our random number example: instead of raising one of two possible exceptions, we'll raise an exception only if we flip a 1.

```
In [25]: import random  # For generating random exceptions.
         num = random.randint(0, 1)
         try:
             if num == 1:
                 raise NameError("This happens when you use a variable you haven't defined")
         except:
             print("I caught something!")
         else:
             print("HOORAY! Lucky coin flip!")
```

```
HOORAY! Lucky coin flip!
```

1.4 Review Questions

Some questions to discuss and consider:

1: Go back to the `if` / `elif` / `else` example about student grades. Let's assume, instead of `elif` for the different conditions, you used a bunch of `if` statements, e.g. `if grade >= 90`, `if grade >= 80`, `if grade >= 70`, and so on; effectively, you didn't use `elif` at all, but just used `if`. What would the final output be in this case?

2: We saw that you can add an `else` statement to the end of an exception handling block, which will run code in the event that no exception is raised. Why is this useful? Why not add the code you want to run in the `try` block itself?

3: With respect to error handling, we discussed `try`, `except`, and `else` statements. There is actually one more: `finally`, which executes *no matter what*, regardless of whether an exception occurs or not. Why would this be useful?

4: There's a whole field of "Boolean Algebra" that is not to different from the "regular" algebra you're familiar with. In this sense, rather than floating-point numbers, variables are either `True` or `False`, but everything still works pretty much the same. Take the following equation: `a(a +`

`b)`. Let's say `a = True` and `b = False`. If multiplication is the `and` operator, and addition is the `or` operator, what's the final result?

5: How would you write the following constraint as a Python conditional: $10 < x \leq 20$, and x is even.

1.5 Course Administrivia

- Assignment 1 is due **tomorrow at 11:59:59pm**. Let me know if there are any questions!
- **How is Assignment 2 going?** Getting your fill of loops?

1.6 Additional Resources

1. Matthes, Eric. *Python Crash Course*. 2016. ISBN-13: 978-1593276034
2. Grus, Joel. *Data Science from Scratch*. 2015. ISBN-13: 978-1491901427