Lecture 5

Blocks; Logical Expressions; Logical Connectives; Designing Logical Expressions; If-Elif-Else

1. Blocks

After an `if .... :`, which statements are controlled by that `if`?

Answer: every statement after, until an **unindented** statement is encountered. The group of statements controlled is also called a ***block statement***. Other languages use curly braces to denote beginning and end of blocks -- Python is notable for eschewing this. It's a little weird at first, but it makes good coding style the law, which I think is really great for beginners.

If your `if` statement has an `else`, that has to come immediately after the end of the `if` block. There can be empty lines between `if` block and `else`, but no unindented non-empty lines. See below.

In [ ]:
```
# EXAMPLE 1a: Block Problems
# What's wrong with this?

if 2 > 1:
    print("Hi")
    print("How's it going")
print("Pretty good")

else:
    print("Blah")
```

In an if-else statement, execution returns to "normal" (every statement is executed) after the end of the `else` block; in a plain if statement, execution returns to "normal" at the end of the `if` block.

This can sometimes be confusing if you see consecutive `if` statements. When one `if` follows the conclusion of another, the two should be considered as separate: the second one will execute the same way it would if the first wasn't there.

For example, what does the following produce when the user enters 1? 10? 100?

In [ ]:
```
# EXAMPLE 1b: More Blocks

x = int(input("Enter a number: "))

if x > 20:
    print("A")
    print("B")
else:
    print("C")
print("D")
if x > 5:
    print("E")
    print("F")
if x > 0:
    print("G")
```

To understand this code, you should break it into chunks: lines 5-9, then line 10, then lines 11-13, and finally lines 14-15. First, pretend only lines 5-9 were present -- the code will execute these first in the way you have hopefully come to understand.

Then, line 10 will always execute.

Then, lines 11-13 execute: if x > 5, E and F print, and there is no `else` so nothing happens with these lines if x is less than or equal to 5.

Finally, lines 14-15 execute: if x > 0, G prints, otherwise nothing happens.

2. Logical Expressions

Question: what would we put into `if ....:` if we wanted something to print out only when x is, say, an odd number between 10 and 30?

This kind of question is a little harder than the ones we've dealt with so far. To deal with it, we will need to get familiar with ***logical expressions***, the gatekeepers (traffic directors?) for if-else statements.

Logical expressions are just like ordinary mathematical expressions, except when you evaluate them, the answers you get are not numbers, but rather `True` or `False`. E.g.:

- `3 + 5` evaluates to 8
- `3 < 5` evaluates to `True`

The basic *arithmetic* operators are +, -, *, /. The basic *logical* operators are >, <, >=, <=, ==, != and also `not`, `and`, `or`.

The first four should be mostly self explanatory. The fifth, ==, is the ***equality*** operator. Note the difference between single = and ==:

- `x = 3` *assigns* the value x to the variable 3. It is a complete statement, more than just an expression.
- `x == 3` *tests* whether x and 3 are equal. This is an *expression*, which will evaluate to `True` or `False`, depending on what value x holds. If you write a line in your program that contains *only* `x == 3`, that line won't do much (but if you put `x == 3` after `if`, that could definitely be useful).

!= is the ***inequality*** operator: `x != y` evaluates to `True` if x and y have different values, and `False` if they have the same value.

In [ ]:
```
# EXAMPLE 2a: Logical Expressions

x = 10
y = 20

print("Here's x < 10:", x < 10)
print("Here's x <= 10:", x <= 10)
print("Here's x * 2 == y:", x * 2 == y)
# Note precedence for the last one: * evaluated before ==.


if x != 10:
    print("A")

if x != 11:
    print("B")

# In C++, this type of thing could drive you CRAZY; fortunately
# Python will raise an error for this.
if y = 20:
    print("C")
```

Note that you can use comparisons with `strs` as well. Equality and inequality should be obvious -- just be aware that equality tests *exact* equality -- cases and spaces both matter.

For < and >, `strs` are compared using ASCII lexigraphical order. What? The "lexigraphical" part is easy to understand; that just means like the dictionary order. You compare the first characters; if they are the same, you go on to the second characters; and so on.

But how would you compare "!23" and "abc"? Recall that ASCII (American Standard Code for Information Interchange) associates each standard character with a number. For instance:

"A" corresponds to 65
"B" corresponds to 66
"a" corresponds to 97
"b" corresponds to 98
"0" (the symbol) corresponds to 49
"!" corresponds to 33

In [ ]:
```
# EXAMPLE 2b: String comparisons

if "yellow" == "Yellow":
    print("A")

if "yellow" == "yellow":
    print("B")

if "a " == "a":
    print("C")

if "2345678" < "239":
    print("D")

if "Baby" < "apple":
    print("E")

if "!23" >= "abc":
    print("F")
```

Choose a password, and the write the following program: it should ask the user to enter a guess; if they enter the correct password, print out a secret message, and otherwise print `Access denied`.

In [ ]:
```
# EXAMPLE 2c: Password
# Ask for input: if the user enters your chosen message, print out secret files.

true_password = "hamburger"
entry = input("Enter password: ")

if entry == true_password:
    print("SECRET FILES")
else:
    print("ACCESS DENIED")
```

Notice my choice of variable names here. There are two "passwords": the true one, and the user's attempt. It's very easy to get confused if you choose less-than-precise variable names (and believe me, situations like this one do happen frequently). So, treat your variable names with respect!

A warning about testing `floats` for equality: umm, avoid doing so. Remember how `floats` are imprecise? Well, tiny imprecisions are one thing, but the difference between `True` and `False` is a bit more stark.

In [ ]:
```
# EXAMPLE 2d: Float Trouble
# The if block ought to execute, but: that's not what happens

if 0.1 + 0.2 == 0.3:
    print("Math is ok")
else:
    print( 0.1 + 0.2 - 0.3)
```

3. Logical Connectives

The other three logical operators, as mentioned, are `not`, `and`, `or`. They connect to other logical expressions, and work as follows.

If `<exp1>` and `<exp2>` are logical expressions, then:

- If `<exp1>` is `True`, then `not <exp1>` evaluates to `False`; and vice-versa.
- `<exp1> and <exp2>` evaluates to `True` if **both** `<exp1>` and `<exp2>` are `True`; otherwise, `<exp1> and <exp2>` evaluates to `False`.
- `<exp1> or <exp2>` evaluates to `True` if **one or both** of `<exp1>` and `<exp2>` are `True`; otherwise (when both are `False`), `<exp1> or <exp2>` evaluates to `False`.

Let's play with some examples. Suppose that `x = 2`. What would the following three expressions evaluate to?

In [ ]:
```
# EXAMPLE 3a: Connectives
# Add in the x = 2 part where you are ready


print(  x > 1 and x < 1 )

print(  not( x > 1 and (x <= 1 or 3 > x) )  )

print(  (3 > x) or (not x > 1) and (x <= 1)  )
```

The first one: `True and False` --> `False`

The second one: `not( True and (False or True) )` --> `not( True and True )` --> `not (True)` --> `False`

The third one: `True or (not True) and False` --> `True or False and False` -->

At this point, we have kind of a trick -- to answer it properly, you need to know more about the rules of precedence! They read, in part:

`**`, then {`*`, `/`, `%`, `//`}, then {`+`, `-`}, then {`==`, `!=`, `<`, `>`, `<=`, `>=`}, then `not`, then `and`, and finally `or`.

Within each class, operators are evaluated as they are encounted from left to right in the expression.

So: `True or False and False` --> `True or False` --> `True`

The `and` is evaluated first!

By the way: `3 > x or not x > 1 and x <= 1` would evaluate the same way, because of those precedence rules: the parentheses are not necessary! But why make your code unreadable? Use parentheses even if they aren't strictly necessary!

One more: if `x = 2`, what would

`1 < x < 3`

evaluate to? There's 3 levels to this question.

1. Everyday mathematics suggest that this is `True`. Duh.
2. But programming languages don't always use everyday reasoning. Let's evaluate this like we did with the expressions above:

   `1 < x < 3` --> `True < 3` ?????

   That's nonsense!
3. But Python is so smart that it realizes that you probably didn't mean for this statement to be interpreted as in level 2, and so it overrides those rules, and instead interprets the expression like in level 1.

I bring all this up because while Python is smart enough to get to level 3 (smart interpretation of *chained inequalities*), C, C++, and Java among others **do not support this.**

4. Designing Logical Expressions

How would you print `You can ride` if the variable `height` is between 36 and 96 (inclusive)?

In [ ]:
```
# EXAMPLE 4a: You can ride

height = 40

if (height >= 36) and (height <= 96):
    print("You can ride.")
```

How would you print `Someone got a 7!` if at least one of `x` and `y` is `7`?

In [ ]:
```
# EXAMPLE 4b: At least one 7

x = 7
y = 5

if (x == 7) or (y == 7):
    print("Someone got a 7!")
```

Important: **note that**

`x or y == 7`

is wrong!

Why? Because think about how this is evaluated: suppose that `x = 7` and `y = 5`. Then first, `y == 5` evaluates to `False`. Then, you are left with evaluating `x or False`. The problem is that `x` is not a logical expression, it's a numerical one!

In general, **both the left and right sides of `or` and and should be *logical* expressions!**

How would you print `Exactly one winner!` if **exactly one** of `x` and `y` is `7`?

In [ ]:
```
# EXAMPLE 4c: Exactly one 7
# Of course, there are several ways to do this

x = 5
y = 7

if (x == 7 or y == 7) and not(x == 7 and y == 7):
    print("Exactly one winner!")

# Alternatively

if (x == 7 and y != 7) or (x != 7 and y == 7):
    print("Exactly one winner!")
```

How would you print `Buzz` if a variable `x` is either a multiple of `7` or ends with the digit `7`?

In [ ]:
```
# EXAMPLE 4d: Buzz

x = 17

if (x % 7 == 0) or (x % 10 == 7):
    print("Buzz")
```

How would you write code which stores the absolute value of `x` into the variable `y`? (Actually, there's a function for that in the `math` library, but let's use `if` to do it instead.)

In [ ]:
```
# EXAMPLE 4e: Absolute value

x = -3

if x >= 0:
    y = x
else:
    y = -x

print(y)
```

5. If-Elif-Else Chains

If-else is great when you have two possibilites to choose from. What if you have 3 possibilities, or 5? In that case, you might want an ***if-elif-else chain***. ("Elif" is short for "else if".) Example:

In [ ]:
```
# EXAMPLE 5a: Pie

favorite = input("Please state your favorite type of pie: ")

if favorite == "Key Lime":
    print("Yeah, that's what's up!")
elif favorite == "Pumpkin" or favorite == "Apple":
    print("That's very American.  I can respect that.")
elif favorite == "Chocolate":
    print("You're not really a pie person, are you?")
else:
    print("Well, at least you didn't put \"Chocolate\".")
```

Here is the general syntax:

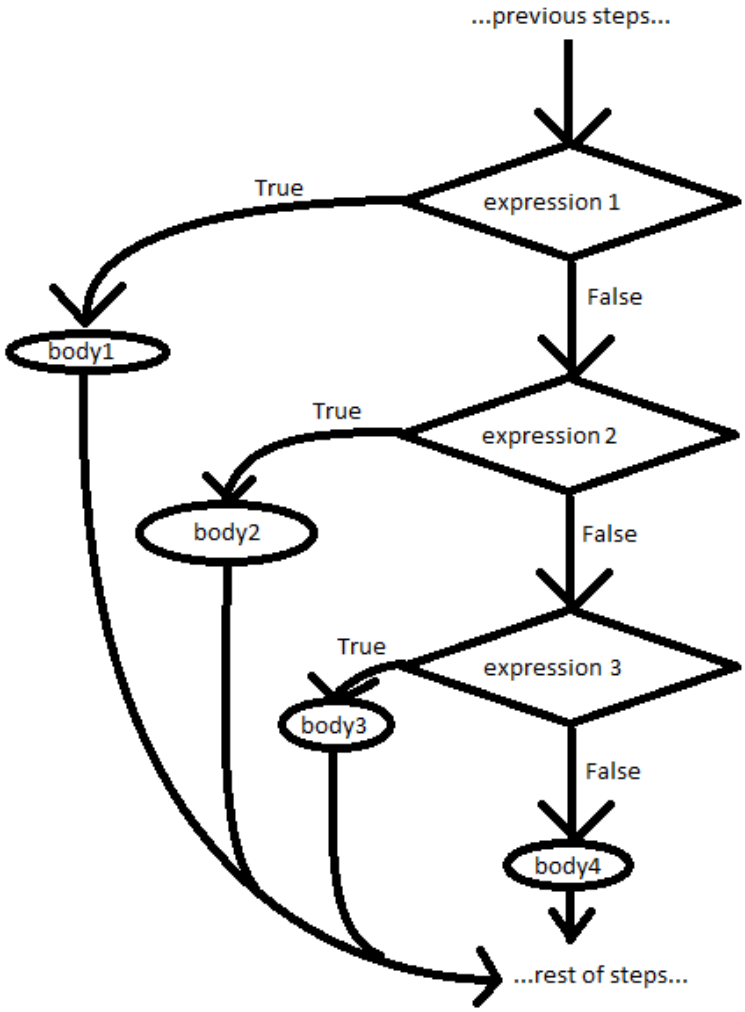
In [ ]:
```
IF-ELIF-ELSE CHAIN SYNTAX:

"...previous statements (unindented)..."

if <expr 1>:
    <body 1>
elif <expr 2>:
    <body 2>
elif <expr 3>:
    <body 3>
else:
    <body 4>

"...further statements (unindented)..."
```

Note that the `elif`'s go in the middle, and of course you can have more or less than 2. Also, the `else` at the end is optional.

The idea is that when this chain is encountered:

- The first logical expression, `<expr 1>`, is evaluated.
- If `True`, `<body 1>` is evaluated, and then execution skips past the rest of the chain.
- If `False`, then execution moves on to the first `elif`, and `<expr 2>` is evaluated. If this is `True`, then `<body 2>` is evaluated; if `False`, execution moves to the third `elif`.
- And so on. If there is an `else` at the end, that executes if none of the logical expressions are `True` (the "default" scenario); if there is no `else` at the end, then it is possible for none of the bodies to execute.

Here's a flowchart, for an if-elif-else chain with an else at the end:



Again: notice that **exactly one** of the bodies is executed. (And if there were no `else`, then **at most one** body would be executed.)

Write a program which asks the user to put in a lowercase letter. Then have it print out `Always` if it is a,e,i,o,u; `Sometimes` if it is y; and `Never` otherwise. (To test this properly, you need a bare minimum of three test runs.)

In [ ]:
```
# EXAMPLE 5b: Vowels
# Ask user to input a lowercase letter, then print out whether or not it is
# a vowel always, sometimes, or never

letter = input("Enter a lowercase letter: ")

if letter == 'a' or letter == 'e' or letter == 'i' or letter == 'o' or letter == 'u
':
    print("Always")
elif letter == 'y':
    print("Sometimes")
else:
    print("Never")
```

I have here three versions of a program. The program is a (slightly lazy) grade converter. It asks for a score, and then prints out `A` for 90-100, `B` for in the 80's, a `C or lower` for less than 80. All versions have flaws. What are the flaws, exactly?

In [ ]:
```
# EXAMPLE 5c: Flawed Grades, Version 1

score = input("Enter score: ")
score = float(score)

if score >= 90:
    print("A")
else score >= 80:
    print("B")
else:
    print("C or lower")
```

In [ ]:
```
# EXAMPLE 5d: Flawed Grades, Version 2

score = input("Enter score: ")
score = float(score)


if score < 80:
    print("C or lower")
elif score >= 80:
    print("B")
elif score >= 90:
    print("A")
```

In [ ]:
```
# EXAMPLE 5e: Flawed Grades, Version 3

score = input("Enter score: ")
score = float(score)

if score >= 90:
    print("A")
if score >= 80:
    print("B")
else:
    print("C or lower")
```

Version 1: This is a straight syntax error: there is no such thing as `else <logical expression>`. `else` is never followed by anything but a `:`. Obviously, `elif` was intended.

Version 2: The backwards order isn't inherently a problem, nor is the lack of an `else`. What IS a problem, however, is that a `score` that is `>= 90` will also be `>= 80`, so B will print out, and you will never get to the second `elif`.

Version 3: Notice that the middle statement is `if`, not `elif` -- so the first two lines are severed from the last four! So if `score >= 90`, then A prints out; then, you execute the next if-else statement: for which B will also print out. Here's a flow chart illustrating: