CS 61A Discussion 1
Control and HOF

Quiz...!

Control is what prevents your programs from simply executing *every* line in top-down order

Without control flow, the code on the right would be run as

```
1.  owen, mad = 0b01, 0x01
2.  if owen is mad:
3.  participate_in_class()
4.  else:
5.  participate_in_class()
```

as opposed to actual behavior, where the `else` clause is ignored.

```
1
2
3
4
5
6
7
8
9   owen, mad = 0b01, 0x01
10  if owen is mad:
11      participate_in_class()
12  else:
13      participate_in_class()
14
15
16
17
18
19
20
21
```

Control Structures

IF, ELIF, ELSE

```
if <boolean expression>:
    <do this>
elif <boolean expression>:
    <do THIS>
else:
    <do THIS!!>
```

You can have as many or as few `elif` clauses as you want. You can have at most one `else` clause.

If one of the boolean expressions evaluates to `True`, its indented suite of statements is executed and all of the subsequent `elif/else`'s (if there are any) are skipped.

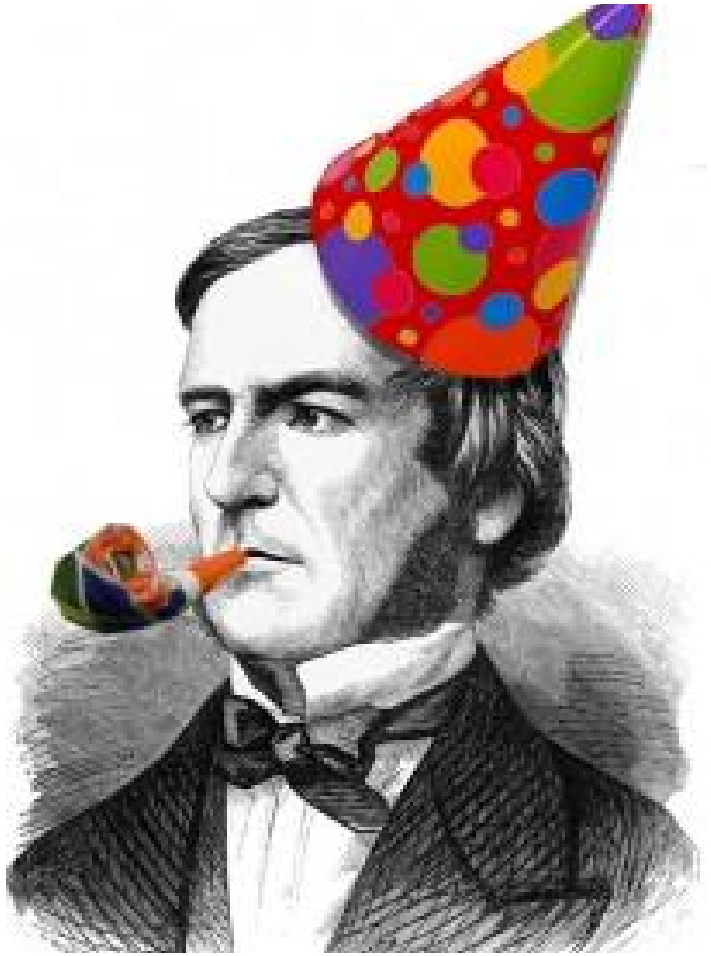
WHILE LOOPS

```
while <boolean expression>:
    <do stuff>
```

When you evaluate a while loop, execution proceeds as follows:

1. Check `<boolean expression>` and see if it is true.
   a. If it isn't, skip all of the lines indented under the `while`. (Continue on with the rest of the program.)
   b. If it IS, *execute* all of the lines indented under the `while`. Then go back to step 1.

Who is this fancy-looking man?



George Boole's Legacy

A **boolean expression** is an expression that is equivalent to either `True` or `False`. This can be any value (or combination of values), because...

...values in Python are either "true"-y or "false"-y.

- False values: `False`, `0`, `[]`, `''`, `None`, `()`, `{}` ("empty" values, basically)
- True values: everything that isn't a false value

You can turn values into more complex boolean expressions by using the `and`, `or`, and `not` operators.

Boolean Operators

And

[PRIORITY LEVEL 2]

True iff *all* of its expressions are true. (False iff *at least one* of its expressions is false.)

Short-circuits if it hits a false value. Always returns the last thing it evaluated.

```
>>> 1 and []
[]
>>> 1 and [3] and 7
7
```

Or

[PRIORITY LEVEL 3]

True iff *at least one* of its expressions is true. (False iff *all* of its expressions are false.)

Short-circuits if it hits a true value. Always returns the last thing it evaluated.

```
>>> 1 or []
1
>>> None or 0 or ()
()
```

Not

[PRIORITY LEVEL 1]

`True` iff its expression is "false"-y. `False` if its expression is "true"-y. (Returns either `True` or `False`.)

```
>>> not (1 and [])
True
>>> not (1 or [])
False
```

Studies show that 50% of people who successfully answer this question also pass the class

What does this evaluate to?

```
>>> a, b, c = 0, 1, 1
>>> b and not a and b - c or not c * a and b / a or b + c
```

Studies show that 50% of people who successfully answer this question also pass the class

What does this evaluate to?

```
>>> a, b, c = 0, 1, 1
>>> b and not a and b - c or not c * a and b / a or b + c
ZeroDivisionError: division by zero
```

1.

Higher Order Functions

Functions as inputs / functions as return values

“You see, in Python functions are first-class values” - John Locke

HOF v1: Functions as Inputs (to Other Functions)

We can pass functions as arguments into function calls. This counts as first-class function manipulation, aka “higher order functions.”

```
>>> from operator import mul
>>> def foo(x, f):
...     return f(f(x, x), f(x, x))
...
>>> foo(5, mul)
625
```

Why do this? Perhaps we want to write a function that is flexible and can perform operations using a broad range of other functions – functions we don’t necessarily know ahead of time.

HOF v2: Functions as Outputs (of Other Functions)

We can also return functions. *Why do this? Perhaps we want the returned function to use values that were part of the original (returning) function, or our program requires a dynamically generated function (something we can use over and over to compute new values).*

```
>>> def foo(x):
...     def bar(y):
...         return not x % y
...     return bar
...
>>> is_factor = foo(400)
>>> is_factor(20)
True
>>> is_factor(21)
False
```

DEMO

Single View Modeling

Thanks for coming!

Until next time...