Cookie Break

I made them, so don’t expect too much.

CS 61A Discussion 1 Control[, Environment Diagrams,] and HOF

THINGS OF POSSIBLE INTEREST

> HW1 has been released. There will be a HW party on Monday from 6:30-8:30pm in 247 Cory.

> MT1 is on Friday 2/17 from 7-9pm.

> The calendar and syllabus have been posted on the website. Please read through the syllabus!

MORE **THINGS**

> The Hog project has been released. There will be project parties next week on Tuesday and Wednesday from 6:30-8:30pm in 247 Cory.

> Water is wet.

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.

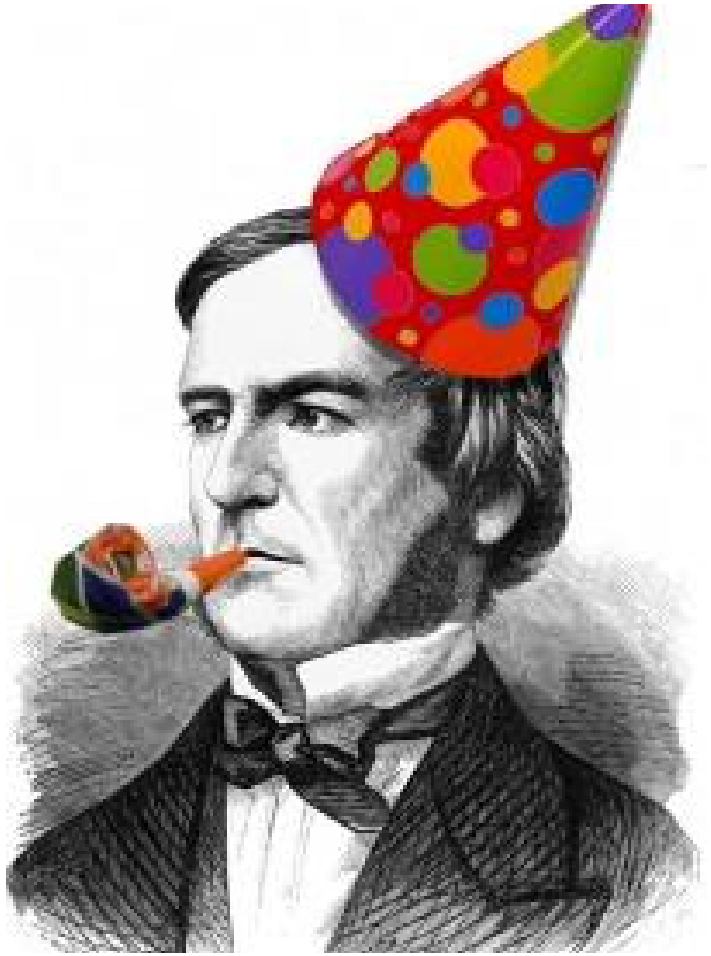
WHILE LOOPS

```
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
```

A Quick Note

"**Priority**" refers to grouping, not evaluation order!

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
```

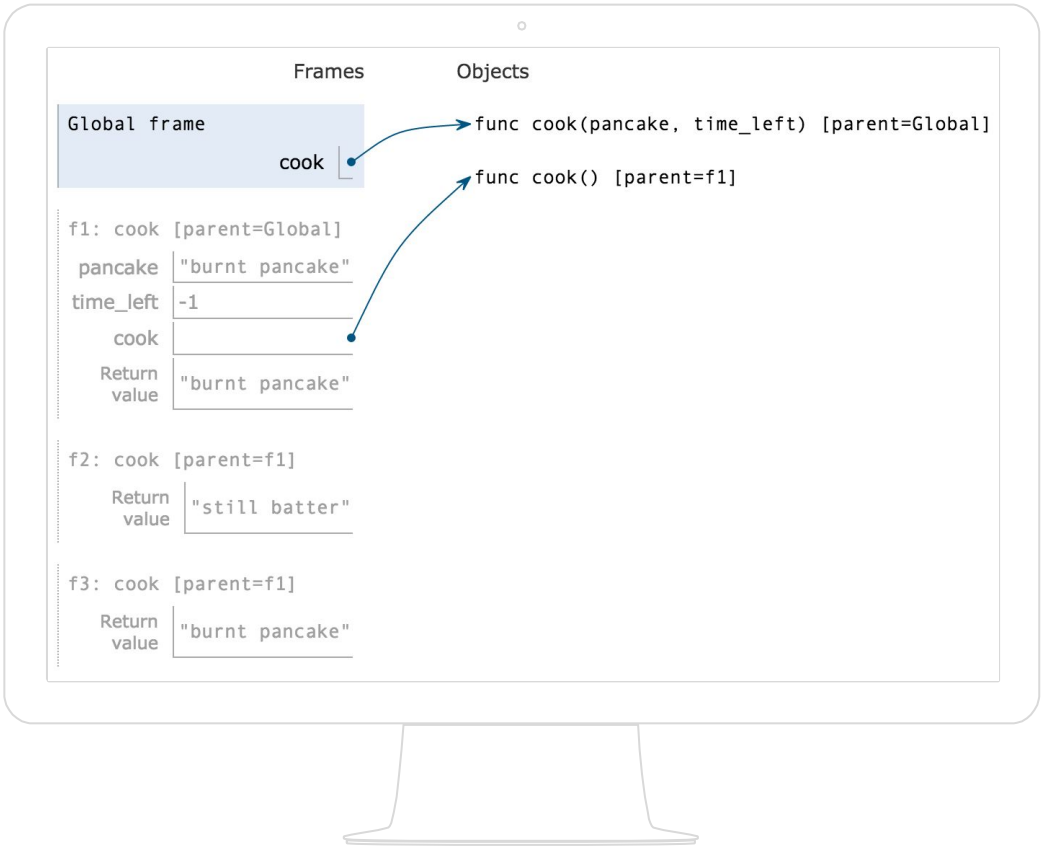
```
* Via priority, the expression is equivalent to
(b and (not a) and (b - c)) or
((not (c * a)) and (b / a)) or
(b + c)
```

Environment Diagrams

should be free points on the exams. You don't have to be creative or clever. You just have to know and follow the rules… albeit mechanically and without error. Still, there *aren't that many rules*! (Hint: they're all in the textbook!)

An environment diagram a day keeps the exam grades okay!

The best way to ensure your free points is through **practice** (as with anything else).



Technically speaking,

an environment diagram is just a record of all name/value bindings.

Drawing a diagram

1. MAKE A GLOBAL FRAME

This is the frame from which all of the other frames are derived.

2. GO THROUGH THE PROGRAM, UPDATING BINDINGS AS NECESSARY

And that's it. Just update them *right*.

ASSIGNMENT STATEMENTS `[x = 5]`

Evaluate the expression(s) on the right-hand side of the = sign. Then bind the value(s) on the right to the name(s) on the left, *in the current frame*.

FUNCTION DEFINITIONS `[def foo(...)]`

Create a new function object, written as `func <fn name>(<params>) [p=<parent>]`. Bind it to the name `<fn name>` in the current frame. The parent is just the current frame.

FUNCTION CALLS `[foo(...)]`

[Step 1: evaluate the operator (the function), then evaluate the operands (the arguments) in order.] Create a new frame. Always! (And *only* when there's a function call, incidentally)! Title the frame `f<curr #>: <intrinsic fn name> [p=<parent>]`. The intrinsic name and the parent **should be obtained from the appropriate function object on the right**. Bind the arguments to the formal parameter names in the newly created frame. Finally, run through the body (also in the new frame). When the function returns, go back to whichever frame you were in before.

LOOKUP `[somevar]`

Check the current frame for the name you're looking for. If it's there, use its value. If not, keep following the chain of parents and doing the same thing. If you get to the global frame and the name still isn't there, it's an error.

(One exception)

FUNCTION CALLS `[foo(...)]`

The only time you don't make a new frame for a function call is when you're calling a **builtin** Python function (`min`, `print`, etc.). We don't know how these are implemented, so we don't add their local frames to our environment diagrams.

If you encounter one of these calls, just assume that it works and continue executing the code.

Common Confusions

- Arguments are evaluated in the *calling* frame
- Everything on the right of an assignment statement is evaluated before the assignment(s) actually happen
- When titling a new frame, use the *intrinsic name* of the function (just copy the name and the parent from the right section of the diagram)
- If you have `x = var`, copy the value of `var` and put it in `x`'s box. If the value of `var` is an arrow pointing to a function, then `x`'s value should be an arrow pointing to the same function.

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
```

Thanks for coming!

Until next time...