

Lecture #4: Higher-Order Functions

Last modified: Wed Jan 29 15:25:21 2014

CS61A: Lecture #4 1

Control

- The expressions we've dealt with recently evaluate *all* of their operands *in order*.
- While there are very clever ways to do everything with just this [challenge!], it's generally clearer to introduce constructs that *control* the order in which their components execute.
- A *control expression* evaluates some or all of its operands in an order depending on the kind of expression, and typically on the values of those operands.
- A *statement* is a construct that produces no value (not even *None*, but is used solely for its side effects).
- A *control statement* is a statement that, like a control expression, evaluates some or all of its components, in an order that may depend on these components.
- We typically speak of statements being *executed* rather than evaluated, but the two concepts are essentially the same, apart from the question of a value.

Last modified: Wed Jan 29 15:25:21 2014

CS61A: Lecture #4 2

Conditional Expressions (I)

- The most common kind of control is *conditional evaluation (execution)*.
- In Python, to evaluate
TruePart if *Condition* else *FalsePart*
 - First evaluate *Condition*.
 - If the result is a "true value," evaluate *TruePart*; its value is then the value of the whole expression.
 - Otherwise, evaluate *FalsePart*; its value is then the value of the whole expression.

• **Example:**

If x is 2:	If x is 0:
$1 / x$ if $x \neq 0$ else 1	$1 / x$ if $x \neq 0$ else 1
$1 / x$ if $2 \neq 0$ else 1	$1 / x$ if $0 \neq 0$ else 1
$\Rightarrow 1 / x$ if True else 1	$\Rightarrow 1 / x$ if False else 1
$\Rightarrow 1 / x$	$\Rightarrow 1$
$\Rightarrow 1 / 2$	$\Rightarrow 1$
$\Rightarrow 0.5$	

Last modified: Wed Jan 29 15:25:21 2014

CS61A: Lecture #4 3

"True Values"

- Conditions in conditional constructs can have any value, not just True or False.
- For convenience, Python treats a number of values as indicating "false":
 - False
 - None
 - 0
 - Empty strings, sets, lists, tuples, and dictionaries.
- All else is a "true value" by default.
- So, for example: 13 if 0 else 5 and 13 if $[]$ else 5 both evaluate to 5.

Last modified: Wed Jan 29 15:25:21 2014

CS61A: Lecture #4 4

Conditional Expressions (II)

- To evaluate
Left and *Right*
 - Evaluate *Left*.
 - If it is a false value, that becomes the value of the whole expression.
 - Otherwise the value of the expression is that of *Right*.
- This is an example of something called "*short-circuit evaluation*."
- For example,
 - 5 and "Hello" \Rightarrow "Hello".
 - 0 and $\text{print}(6)$ $\Rightarrow 0$ + side-effects: None.
 - $[]$ and $1 / 0$ $\Rightarrow []$.

Last modified: Wed Jan 29 15:25:21 2014

CS61A: Lecture #4 5

Conditional Expressions (III)

- To evaluate
Left or *Right*
 - Evaluate *Left*.
 - If it is a true value, that becomes the value of the whole expression.
 - Otherwise the value of the expression is that of *Right*.
- Another example of "*short-circuit evaluation*."
- For example,
 - 5 or "Hello" $\Rightarrow 5$.
 - 2 or $\text{print}(6)$ $\Rightarrow 2$ + side-effects: None.
 - $[]$ or $1 / 0$ \Rightarrow error.

Last modified: Wed Jan 29 15:25:21 2014

CS61A: Lecture #4 6

Chained Comparisons

- An interesting feature of Python (quite rare; Cobol has something like it) involves the relational operators:
`== != < > <= >= is is not in not in`
- Ordinarily, `3<4` yields `True` and `4<3` yields `False`.
- But what does `4 >= 3 > 1` produce? In Java, it's an error, and in C, it doesn't do what you probably want.
- In Python, it's a special control expression and works as expected.
- To evaluate `First > Second >= Third`, for example,
 - Evaluate `First` and `Second`.
 - If the first value is not larger than the second, stop and yield `False` for the entire expression.
 - Otherwise, compute the value of `Third` and compare against the value previously computed for `Second`, and yield `True` or `False` as appropriate.
 - In any case, no expression is evaluated more than once.

Last modified: Wed Jan 29 15:25:21 2014

CS61A: Lecture #4 7

Chained Comparisons (II)

- So what is
`(print("A") or 3) < (print("B") or 2) < (print("C") or 4)`
and what does it print?
- Prints A and B, evaluates to False.

Last modified: Wed Jan 29 15:25:21 2014

CS61A: Lecture #4 8

Conditional Statement

- Finally, this all comes in statement form:

```
if Condition1:
    Statements1
...
elif Condition2:
    Statements2
...
...
else:
    Statementsn
...
```

- Execute (only) `Statements1` if `Condition1` evaluates to a true value.
- Otherwise execute `Statements2` if `Condition2` evaluates to a true value (optional part).
- ...
- Otherwise execute `Statementsn` (optional part).

Last modified: Wed Jan 29 15:25:21 2014

CS61A: Lecture #4 9

Example

```
def signum(x):
    if x > 0:
        return 1
    elif x == 0:
        return 0
    else:
        return -1

# Alternative Definition
def signum(x):
    return 1 if x > 0 else 0 if x == 0 else -1
```

Last modified: Wed Jan 29 15:25:21 2014

CS61A: Lecture #4 10

A Puzzle: Define compare3

What goes here?

```
from operator import lt, gt # Comparison functions
```

```
gt(gt(3,2), 1) # Yields False, not like 3>2>1 (why?)
```

```
compare3(gt)(3)(2)(1) # This should yield True
compare3(gt)(3)(2)(4) # This should yield False
compare3(lt)(1)(2)(3) # This should yield True
# etc.
```

Last modified: Wed Jan 29 15:25:21 2014

CS61A: Lecture #4 11

Some Solutions

```
def compare3(op):
    def f(a):
        def g(b):
            return lambda c: op(a,b) and op(b, c)
        return g
    return f

def compare3(op):
    def f(a):
        def g(b):
            if op(a,b):
                return lambda c: op(b, c)
            else:
                return lambda c: False
        return g
    return f
```

Last modified: Wed Jan 29 15:25:21 2014

CS61A: Lecture #4 12

Indefinite Repetition

- With conditionals and function calls, we can conduct computations of any length.
- For example, to sum the squares of all numbers from 1 to N (a parameter):

```
def sum_squares(N):
    """The sum of K**2 for K from 1 to N (inclusive)."""
    if N < 1:
        return 0
    else:
        return N**2 + sum_squares(N - 1)
```

- This will repeatedly call `sum_squares` with decreasing values (down to 1), adding in squares:

```
sum_squares(3) => 3**2 + sum_squares(2)
=> 3**2 + (2**2 + sum_squares(1))
=> 3**2 + (2**2 + (1**2 + sum_squares(0)))
=> 3**2 + (2**2 + (1**2 + 0)) => 14
```

Last modified: Wed Jan 29 15:25:21 2014

CS61A: Lecture #4 13

Explicit Repetition

- But in the Python, C, Java, and Fortran communities, it is more usual to be explicit about the repetition.

- The simplest form is **while**

`while` *Condition*:
Statements

means "If condition evaluates to a true value, execute statements and repeat the entire process. Otherwise, do nothing."

- So our sum-of-squares becomes:

```
def sum_squares(N):
    """The sum of K**2 for K from 1 to N (inclusive)."""
    result = 0
    while N >= 1:
        result += N**2    # Or result = result + N**2
        N -= 1           # Or N = N-1
    return result
```

- (Actually, this isn't quite right. What's different from the first version?)

Last modified: Wed Jan 29 15:25:21 2014

CS61A: Lecture #4 14

Going Backwards

- OK: I cheated. In the recursive version, you actually add up the squares starting from the small end.
- So to be true to the original, I would write:

```
def sum_squares(N):
    """The sum of K**2 for K from 1 to N (inclusive)."""
    result = 0
    k = 1
    while k <= N:
        result += k**2
        k += 1
    return result
```

Last modified: Wed Jan 29 15:25:21 2014

CS61A: Lecture #4 15

Definite Repetition

- In most programming languages, we write "counting loops" like the preceding with a specialized kind of loop. In Python:

```
def sum_squares(N):
    """The sum of K**2 for K from 1 to N (inclusive)."""
    result = 0
    # Original:
    # k = 1
    # while k <= N:
    #     result += k**2
    #     k += 1
    for k in range(1, N+1):
        result += k**2
    return result
```

- This actually means "execute `result += k**2` for every value of `k` in the range 1 (inclusive) to `N+1` (exclusive)."
- Special case of a more general version that we'll see later.

Last modified: Wed Jan 29 15:25:21 2014

CS61A: Lecture #4 16