Lecture #4: 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 the 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.

Conditional Expressions (I)

- The most common kind of control is conditional evaluation (execution).
- In Python, to evaluate
  \[ \text{TruePart if } Condition \text{ else } \text{FalsePart} \]
  - 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:
    \[ \frac{1}{x} \text{ if } x \neq 0 \text{ else } 1 \]
    \[ \frac{1}{2} \text{ if } True \text{ else } 1 \]
    \[ \frac{1}{2} \]
    \[ 0.5 \]

  - If x is 0:
    \[ \frac{1}{x} \text{ if } x \neq 0 \text{ else } 1 \]
    \[ \frac{1}{x} \text{ if } False \text{ else } 1 \]
    \[ 1 \]

"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.

Conditional Expressions (II)

- To evaluate
  \[ \text{Left and } \text{Right} \]
  - 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,
  \[ 5 \text{ and } "Hello" \]
  \[ "Hello" \]
  \[ 0 \text{ and } \text{print}(6) \]
  \[ 6 \text{ + side-effects: None.} \]
  \[ [] \text{ or } 1/0 \]
  \[ \text{error} \]

Conditional Expressions (III)

- To evaluate
  \[ \text{Left or Right} \]
  - 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,
  \[ 5 \text{ or } "Hello" \]
  \[ [] \]
  \[ 2 \text{ or } \text{print}(6) \]
  \[ [] \text{ + side-effects: None.} \]
  \[ [] \text{ or } 1/0 \]
  \[ \text{error} \]

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.
Chained Comparisons (II)

- So what is
  \[ (\text{print}("A") \text{ or } 3) < (\text{print}("B") \text{ or } 2) < (\text{print}("C") \text{ or } 4) \]
  and what does it print?

- Prints A and B, evaluates to False.

Conditional Statement

- Finally, this all comes in statement form:

  ```python
  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).

Example

```python
# Alternative Definition

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

A Puzzle: Define compare3

```python
# What goes here?

from operator import lt, gt # Comparison functions

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

cmpare3(gt)(3)(2)(1) # This should yield True

cmpare3(gt)(3)(2)(4) # This should yield False

cmpare3(lt)(1)(2)(3) # This should yield True # etc.
```

Some Solutions

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

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):

  ```python
  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
    ```
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?)

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

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.