CS 115 Lecture 9

Boolean logic; random numbers

Neil Moore

Department of Computer Science
University of Kentucky
Lexington, Kentucky 40506
neil@cs.uky.edu

29 September 2015
1 October 2015

Augmented assignment

Often you want to perform an operation on a variable and store the result in the same variable:

```
num_students = num_students + 1
price = price * 0.9  # 10 percent discount
change = change % 25  # change after quarters
```

Python provides a shorthand for this, **augmented assignment**:

```
num_students += 1
price *= 0.9
change %= 25
```

- Combines assignment with an arithmetic operator.
- The precedence is the same as assignment (=).
    - Evaluate the right hand side first.
    - What does this do? `product *= i + 1`
    - Not: `product = product * i + 1`
    - But: `product = product * (i + 1)`

Comparing strings

The relational operators <, >=, etc. work with strings, too.

- Uses a form of "lexicographic" (alphabetical, dictionary) order.
  - Compare corresponding characters in order.
  - The first difference tells us the answer.
  - `'comparison' < 'compiler'`
  - Prefix comes "first": `'pick' < 'pickle'`
- Compares the numeric code (**Unicode**) for each character.
  - Mostly alphabetic for basic English characters.
  - Uppercase before lowercase! `'Z' < 'a'`
  - Digits come before letters. `'A2' < 'AA'`
  - Space comes before digits and letters. `'good day' < 'goodbye'`
  - `' '` $<$ `'0'` $< \cdots <$ `'9'` $<$ `'A'` $< \cdots <$ `'Z'` $<$ `'a'` $< \cdots <$ `'z'`
  - **ASCII** is a subset of Unicode with only basic English characters. `https://en.wikipedia.org/wiki/ASCII#ASCII_printable_code_chart`
- Can't compare a string to a number, only to other strings!

Chaining comparisons

- In Python, comparisons can be chained together:
  ```
  if 0 < x < y <= 100:
  ```
- Means: $0 < x$ and $x < y$ and $y \leq 100$.
- This notation is common in mathematics.
  - But not in most programming languages!
  - Python is rather unique in allowing it.

Boolean logic

There are three **logical operators** that let us combine boolean expressions. They have lower precedence than the relational operators.

- `not A`: True if **A** is false, false if **A** is true.
  - `A` can be any boolean expression:

    ```
    if not is_finished:
        do_more_work()
    ```
- `A and B`: True if **both** `A` and `B` are true.

    ```
    in_range = size >= 0 and size <= 100
    ```
- `A or B`: True if **either** `A` or `B` is true.
  - Or both!

    ```
    if snow_inches > 6 or temperature < 0:
        print("Class is cancelled")
    ```

Complex boolean expressions

- `not` has highest precedence (still lower than comparison).
- `and` has the next highest.
- `or` has the lowest.
- So `not A or B and C or D` means:
  `(((not A) or (B and C)) or D)`
- People often forget the order of `and` and `or`
  - It's not a bad idea to always use parentheses when combining them.
    `not A or (B and C) or D`

Truth tables

The **truth table** is a tool for making sense of complex boolean expressions.

A	not A
T	F
F	T

A	B	A and B
T	T	T
T	F	F
F	T	F
F	F	F

A	B	A or B
T	T	T
T	F	T
F	T	T
F	F	F

- One row for each possible combination of values
  - If there is one input, two rows (T, F).
  - Two inputs, four rows (TT, TF, FT, FF).
  - 3 inputs, 8 rows (TTT, TTF, TFT, TFF, FTT, FTF, FFT, FFF).
- A column for each boolean expression.
  - Inputs: Boolean variables, comparisons (relational expressions).
  - Intermediates: Each **not**, **and**, and **or**.
  - Output: The whole expression.

A more complicated example

`not (not A or not B)`

A	B	not A	not B	not A or not B	answer
T	T	F	F	F	T
T	F	F	T	T	F
F	T	T	F	T	F
F	F	T	T	T	F

De Morgan's laws:

- not (not A or not B) = A and B
- not (not A and not B) = A or B

Be careful!

It is easy to accidentally write an expression that is *always* true, or *always* false.

- **Tautology** and **contradiction**.
- An example:

```
if size >= 10 or size < 50:
    print("in range")
```

  - What happens when size is 100? 20? 2?
  - `or` is true if either comparison is true.
  - But they can't ever both be false!
  - So this `or` is always true (a tautology).

```
if size < 10 and size > 100:
    print("out of range")
```

  - The comparisons can't ever both be true!
  - A contradiction—will never print the message.

Be careful!

- Don't trust the English language!
  - Make a truth table if you are unsure.
- "I want to run this if size $<$ 10 and if size $>$ 100"
  - In logic, that's an **or**, not an and:
    "Run this if size $<$ 10 or size $>$ 100"
  - (The example from last slide)
- "If x is equal to 4 or 5. . . "
  - Wrong: `if x == 4 or 5:`
  - Boolean expressions are like sentences.
    - ★ But here "or" joins nouns, not sentences.
  - Instead: "If x is equal to 4 or x is equal to 5"
    `if x == 4 or x == 5:`

Python modules

We've already seen a couple of **modules** or **libraries** in Python:

- `math`
- `graphics`
- A collection of pre-written code intended to be re-used.
- Python comes with a couple *hundred* modules.
- And there are thousands mode third-party modules.
- Let's look at one more: `random`

Randomness

The `random` module provides functions for generating **random numbers**.

- Computers are **deterministic**:
    - The same instructions and the same data = the same results.
    - Usually this is what we want.
    - When might we want the program to do a different thing every time?
        - Games.
        - Simulations: traffic, weather, galaxies colliding, ...
        - Cryptography.
- For these kinds of problems we want **random numbers**.
    - But how can we get real randomness in a deterministic machine?
    - There are ways, but usually it's not necessary.
    - **Pseudorandom** numbers are usually good enough.

Randomness

What does "random" mean? Two things:

- An even distribution of results.
  - If we're rolling a die, we expect 1 about 1/6 of the time.
  - and 2 about 1/6 of the time, 3 about 1/6, . . .
  - **Uniform distribution**: each possibility is equally likely
  - This *doesn't* mean exactly uniform results!
    - ⋆ Roll a die six times: I bet you get some number twice.
    - ⋆ Over a large number of tests, gets closer to 1/6 each.
- An even distribution isn't enough to be "random"
  - What if the die always rolled 1, 2, 3, 4, 5, 6 in that order?
  - Random numbers should be **unpredictable**.
  - Specifically, seeing several numbers shouldn't let us guess the next one.

Psseudorandom numbers

Pseudorandom numbers use a deterministic procedure (a **random number generator**, RNG) to generate numbers that appear to be random:

- Approximately uniform.
- Hard to predict (maybe not impossible).
  - RNGs will repeat eventually: want this to take a long time.
- A lot of research has gone (and goes) into RNGs:
  - Linear congruential, alternating shift generator, Mersenne twister, . . .
  - *The Art of Computer Programming* spends half a book on RNGs.
  - Why so much research? Important for security!
    - ⋆ Cryptography uses random numbers for **session keys** (like automatically generated one-time passwords).
    - ⋆ If someone could predict the output of the RNG, they could predict the key and break in or read your data!

Using Python's random number library

Python's random number generator is in the `random` library.

- `import random` or `from random import *`
- There are several functions in the library.
  - `https://docs.python.org/3/library/random.html`
  - (Note the big red warning!)
- The simplest function is `random`:

  ```
  chance = random()
  ```

  - Gives a random float in the range $[0.0, 1.0)$:
    - Notation: including 0.0, not including 1.0.
  - Useful for probabilities: 1 means "will happen", 0 means "will not"

    ```
    if random() < 0.7:  # 70% chance
    ```

- What if we want a random float in a different range?
  - Multiply and add:

    ```
    score = 90.0 * random() + 10.0
    ```

    Range: $[10.0, 100.0)$

Random integers

We could multiply, add, and type-cast to get a random integer.
But there's a simpler and better way.

- The `randrange` function.
- Takes one to three arguments and returns an integer:
  - `randrange(stop)`: $[0, stop)$
    Between zero (inclusive) and `stop` (*exclusive*!)
  - `randrange(start, stop)`: $[start, stop)$
    Between start (inclusive) and `stop` (exclusive)
  - `randrange(start, stop, step)`:
    Likewise, but only gives `start` plus a multiple of `step`.
- "Give me a random multiple of 10 between 0 and 100 inclusive."
  - `score = randrange(0, 101, 10)`
  - What if we wrote 100 instead? Wouldn't be inclusive.
- Related: `randint(a, b)`: $[a, b]$
  - Inclusive on both ends! The same as `randrange(a, b + 1)`
  - Prefer `randrange` in new code.

Random choice

Python can also choose randomly from a list of alternatives:

```
sacrifice = choice(["time", "money", "quality"])
```

- Must give a **list** of choices, in square brackets.
  - Don't forget the brackets!

    ```
    choice("time", "money", "quality")
      → TypeError: choice() takes 2 positional arguments...
    ```
- Can give a string instead: `answer = choice("ABCD")`
  Returns a random letter from the string.

Seeding the RNG

Sometimes it's useful to be able to repeat the program exactly, with the same sequence of random numbers. Why?

- Reproducible simulations.
- Cryptography: client and server might need the same numbers.
- Testing games (and "tool-assisted speedruns").
- We can specify the **seed** for the RNG.
  - `seed(42)` — once at the beginning of the program.
  - Now the sequence of numbers will be the same each time.
  - `seed(43)`: completely different sequence.
    - Not necessarily larger numbers!
- What if you never set a seed?
  - Python picks one for you, based on the system time.
  - On some OSes it can use OS randomness instead.
- Only set the seed once per program!