CS 115 Lecture

Random numbers

Taken from notes by Dr. Neil Moore

Random numbers

- We've seen some modules or libraries in Python:
  - `math`
  - `graphics`
  - A library is a collection of pre-written code indented to be re-used.
  - Python comes with a couple *hundred* modules
  - And there are thousands more third-party modules
  - Let's look at another built-in module: `random`

Randomness

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

- Computers are **deterministic**:
  - The same instructions given the same data (input) yields the same results every time
  - Usually this is what we want!
  - When would we want a program to do different things every time it's run?
    - Games
    - Simulations of the real world: 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 (hooking up a radiation source and look for decays, etc. …) but it's not needed most of the time
  - **Pseudorandom** numbers are usually good enough for our purposes

Randomness

What does “random” mean?

- An even distribution of results
  - If we’re rolling a die, we expect 1 about $1/6^{th}$ of the time
  - And 2 about $1/6^{th}$ of the time, 3 about $1/6^{th}$ …
  - **Uniform distribution**: each possibility is equally likely
  - This does NOT mean exactly uniform results!
    - If you roll a die six times, you will get some number twice
  - What it means is that **over a large number of tests**, the distribution gets closer to $1/6^{th}$ each
- An even distribution isn’t enough to be “random”
  - What if the die always rolled 1,2,3,4,5,6, 1,2,3,4,5,6,… in that order?
  - Random numbers should be **unpredictable**
  - Specifically seeing several numbers in the series should not let us guess the next one

Pseudorandom numbers

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

- Approximately uniform
- Hard to predict (but theoretically not impossible)
  - ALL RNGs will repeat eventually, a good one does not for a very 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? They are very important for security!
    - Cryptograph 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!

Randomness involves information

- Randomness involves information or the lack of it
- Imagine you are standing at the top of a 50-story building.
- If someone asked you to predict what the traffic at ground-level would be, “when will the next car come around the corner?” you would be in a good position to make a prediction because you can see the streets for a long way
- If you were standing at ground level next to the same building, you have much less information and you could not make a very good prediction about the traffic
- With **more** information, things are **less** random; with **less** information, things seem **more** random
- That’s why the RNG numbers are called **pseudo**. With enough information, i.e. the RNG algorithm used and the **seed**, you could calculate the numbers just like the computer does. The numbers ARE predictable in that sense. Since we don’t usually have that info (or want to do that), the numbers seem random to us!

A good RNG

- What makes a “good” random number generator?
  - The same features we mentioned earlier – uniform distribution, being unpredictable
  - It must be quick to calculate – a typical game would use millions of them
    - Most of them use integer arithmetic because it’s faster than floating point
  - It must have a long cycle before it repeats
    - EVERY RNG will eventually repeat if run long enough, but if the cycle is a few million numbers, it will seem “unpredictable” for most humans
      - Why will it repeat? Imagine you had a “perfect” RNG which would generate every possible number the computer could represent (in no predictable order). That’s a finite number of numbers. What if you asked for **another** random number after it had done that? It HAS to give you a number it has already given before! It has no others to give.

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! This RNG should not be used for critical applications like banking!
- The simplest function is random

  ```
  chance = random()
  ```
  - Gives a random floating point number 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 happen"

  ```
  if random() < 0.7:  # 70% chance to be True
  ```
- What if we want a random float in another range?
  - You could multiply and add

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

    the range of this variable is [10.0, 100.0)

Using random()

- We want the program to print out a greeting
- 30% of the time it should be “hi!”
- 20% of the time it should be “hello”
- And 50% of the time it should be “How are you?”
- rnd = random() # gives a float number between 0 and 1
- if rnd <= 0.3:

```
        print(“hi!”)
elif rnd <= 0.5:    #between 0.3 and 0.5, a space of 0.2
        print(“hello”)
else:
        print(“How are you?”)
```

Random integers

We could multiply, add and type-cast to get a random integer from the random function. But there are functions in the library to do it for us.

- The `randrange` function
- Takes one to three arguments and returns an integer:
  - `randrange(stop)`:

    between zero (inclusive) and `stop` (*exclusive*! Not including `stop`)
  - `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`

Random integers

- “Give me a random multiple of 10 between 0 and 100 inclusive.”
  - score = randrange(0, 101, 10)
  - What if we had written 100 instead? 100 is not included in the possible results
- Also: `randint (a, b):`
  - Inclusive on both ends! The same as `randrange (a, b+1)`
  - Always has two arguments, no more, no less
  - Returns numbers in a range starting at `a`, up to and including `b`

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(2) takes 2 positional arguments but four were given
```

- Can give a string as an argument instead: `answer=choice("ABCD")`

  Returns a random letter from the string

  Could get the same result with `answer = "ABCD"[randrange(4)]`

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 programs (and games)
- We can specify the **seed** for the RNG
  - `seed(42)` do it ONCE at the beginning of the program
  - Now the sequence of numbers will be the same each time the program runs
  - `seed(43)` gives a completely different random number sequence
    - Not necessarily larger numbers (size of seed does not correlate with size of 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!