

# ENGG1003 - Monday Week 4

Iteration again: `for` vs. `while` loops,  
debugging strategies & random numbers

Steve Weller

University of Newcastle

15 March, 2021

# Lecture overview

- 1 Iteration again (again): `for` vs. `while` loops  
§3.3.3
- 2 Debugging strategies
- 3 Random numbers in Python §2.4

# 1) iteration again: `for` vs. `while` loops

- side-by-side comparison for print-1-to-10
- similarities and differences
- when to use each

● XXX

# Example: Finding the maximum height

- §3.3.3
- new program instead finds the maximum height achieved by the ball
- will solve in two ways: for and while

```

import numpy as np
import matplotlib.pyplot as plt

v0 = 5                                # Initial velocity
g = 9.81                              # Acceleration of gravity
t = np.linspace(0, 1, 1000)          # 1000 points in time interval
y = v0*t - 0.5*g*t**2                # Generate all heights

# At this point, the array y with all the heights is ready,
# and we need to find the largest value within y.

largest_height = y[0]                 # Starting value for search
for i in range(1, len(y), 1):
    if y[i] > largest_height:
        largest_height = y[i]

print('The largest height achieved was {:.g} m'.format(largest_height))

# We might also like to plot the path again just to compare
plt.plot(t,y)
plt.xlabel('Time (s)')
plt.ylabel('Height (m)')
plt.show()

```

# Focus: `for` loop to find max height

- describe strategy in words
- live demo

```
largest_height = y[0]           # Starting value for search
for i in range(1, len(y), 1):
    if y[i] > largest_height:
        largest_height = y[i]
```

The largest height achieved was 1.27421 m

# Focus: `while` loop to find max height

- describe strategy in words
- live demo

```
i = 0
while y[i+1] > y[i]:
    i = i + 1
```



## 2) Debugging strategies

- 1 running code by hand
- 2 don't guess, print!
- 3 take baby steps

# Running code by hand

- Work through your code line-by-line, with a piece of paper and a pen
- Use paper/notes-app before you run code, so that you know what you want program to calculate on each line before you run code
- Second-best: use notes app on iPad or similar—idea is to *think before computing*
- Check that every line of code, with every use of an array index (hint) matches what you expect
- Near enough isn't good enough here. Think like a computer: work systematically through each line of code. Is it doing what you want it to do?

# Don't guess, print!

- XXX

# Take baby steps

- XXX

### 3) Random numbers in Python

- Python provides ability to produce (apparently) random numbers
- referred to as *pseudo-random numbers*
- these numbers are not *truly* random
  - ▶ produced in a complicated (but “deterministic” or predictable) way once a *seed* has been set
- seed is a number which depends on the current time

# Drawing **one** random number at a time

```
import random

a = 1; b = 6
r1 = random.randint(a, b)    # first die
r2 = random.randint(a, b)    # second die

print('The dice gave:  {:d} and {:d}'.format(r1, r2))
```

Python code: throw\_2\_dice.py

- function `randint(a, b)`
  - ▶ available from imported module `random`
  - ▶ returns a pseudo-random *integer* in the range  $[a, b]$  where  $a \leq b$

# Fixing the seed

- when debugging programs that involve pseudo-random numbers, often helps to *fix the seed*
- ensures that *identical sequence of numbers will be generated* each time code is run
  - ▶ hence results are *repeatable*
- tell Python what seed should be using `random.seed` function
- **Example:** `random.seed(10)` and run Python code: `throw_2_dice.py`

# Two functions: `random` and `uniform`

- both `random` and `uniform` return a floating point number from an interval where each number has *equal probability* of being drawn
  - ▶ random number drawn from *uniform* probability distribution
  - ▶ Note: `random` function in `random` module
- `random`
  - ▶ draw from interval  $[0, 1)$
- `uniform`
  - ▶ draw from interval  $[a, b]$



# Live demo: random and uniform

```
In [1]: import random
```

```
In [2]: x = random.random()           # draw float from [0, 1), assign to x
```

```
In [3]: y = random.uniform(10, 20) # ...float from [10, 20], assign to y
```

```
In [4]: print('x = {:g}, y = {:g}'.format(x, y))
```

```
Out[5]: x = 0.714621 , y = 13.1233
```

# Drawing **many** random numbers at a time

- three random number generators seen so far
  - ▶ each generates just *one* random number at a time
- to generate an *array* of random numbers...  
...could use a loop & generate one random number in each iteration
- better (faster) solution: use `random` module in `numpy` library

# Live demo: random numbers from numpy library

**Example:** `np.random.randint`

- ▶ numpy library / random module / randint function
- ▶ `randint(a,b,n)` generates  $n$  integers from  $[a,b)$

```
In [1]: import numpy as np
```

```
In [2]: np.random.randint(1, 6, 4)      # ...4 integers from [1, 6)
```

```
Out[2]: array([1, 3, 5, 3])
```

```
In [3]: np.random.random(4)            # ...4 floats from [0, 1)
```

```
Out[3]: array([ 0.79183276,  0.01398365,  0.04982849,  0.11630963])
```

```
In [4]: np.random.uniform(10, 20, 4)   # ...4 floats from [10, 20)
```

```
Out[4]: array([ 10.95846078,  17.3971301 ,  19.73964488,  18.14332234])
```

- live demo, also fix seed: `np.random.seed(10)`

# Lecture summary

- Iteration again
  - ▶ `for` vs. `while`
- Debugging strategies
  - ▶ running code by hand
  - ▶ don't guess, print!
  - ▶ take baby steps
- Random numbers
  - ▶ `random` module—random numbers one at a time
  - ▶ `random` module in `numpy` library—arrays of random numbers