ENGG1003 - Monday Week 3

Loops and branching

Steve Weller

University of Newcastle

8 March, 2021

Last compiled: March 6, 2021 2:31pm +11:00

Lecture overview

1. Iteration using `for` loop §3.1
   - fixed number of iterations
2. Iteration using `while` loop §3.2
   - keep iterating whenever a condition is satisfied
3. Branching: `if`, `elif` and `else` §3.3
   - check condition before executing code block

1) Iteration using `for` loop

- many computations are repetitive by nature and programming languages have certain loop structures to deal with this
- one such loop structure is the for loop
- printing the 5 times table
- **at console—begin with live demo**

```python
# Naively printing the 5 times table
print('{:d}*5 = {:d}'.format(1, 1*5))
print('{:d}*5 = {:d}'.format(2, 2*5))
print('{:d}*5 = {:d}'.format(3, 3*5))
print('{:d}*5 = {:d}'.format(4, 4*5))
print('{:d}*5 = {:d}'.format(5, 5*5))
print('{:d}*5 = {:d}'.format(6, 6*5))
print('{:d}*5 = {:d}'.format(7, 7*5))
print('{:d}*5 = {:d}'.format(8, 8*5))
print('{:d}*5 = {:d}'.format(9, 9*5))
print('{:d}*5 = {:d}'.format(10, 10*5))
```

```
1*5  = 5
2*5  = 10
...
...
```

- first loop for i in [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]:
- code fragment LLp60

```
for i in [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]:     # Note... for, in and colon
    print('{:d}*5 = {:d}'.format(i, i*5))         # Note indent
```

A typical for loop

- general loop structure
- indentation—critical!

```
for loop_variable in some_numbers:   # Loop header
    <code line 1>                        # 1st line in loop body
    <code line 2>                        # 2nd line in loop body
    ...
    ...                              # last line in loop body
# First line after the loop
```

Nested loops

```python
for i in [1, 2, 3]:
    # First indentation level (4 spaces)
    print('i = {:d}'.format(i))
    for j in [4.0, 5.0, 6.0]:
        # Second indentation level (4+4 spaces)
        print('      j = {:.1f}'.format(j))
    # First line AFTER loop over j
# First line AFTER loop over i
```

```
i = 1
      j = 4.0
      j = 5.0
      j = 6.0
i = 2
      j = 4.0
      j = 5.0
      j = 6.0
i = 3
      j = 4.0
      j = 5.0
      j = 6.0
```

Combining for loop and array

- average height
- §3.1.3

```
import numpy as np

N = 5
h = np.zeros(N)     # heights of family members (in meter)
h[0] = 1.60; h[1] = 1.85; h[2] = 1.75; h[3] = 1.80; h[4] = 0.50

sum = 0
for i in [0, 1, 2, 3, 4]:
    sum = sum + h[i]
average = sum/N

print('Average height: {:g} meter'.format(average))
```

range function

- motivation:
- range(start, stop, step)
- eg: range(0,5,1)
- why tf the weird indexing?

```
for i in [0, 1, 2, 3, 4]:       # original code line
```

```
for i in range(0, 5, 1):        # new code line
```

Live demo: for loop

- Python code: `average_height.py`

2) Iteration using `while` loop

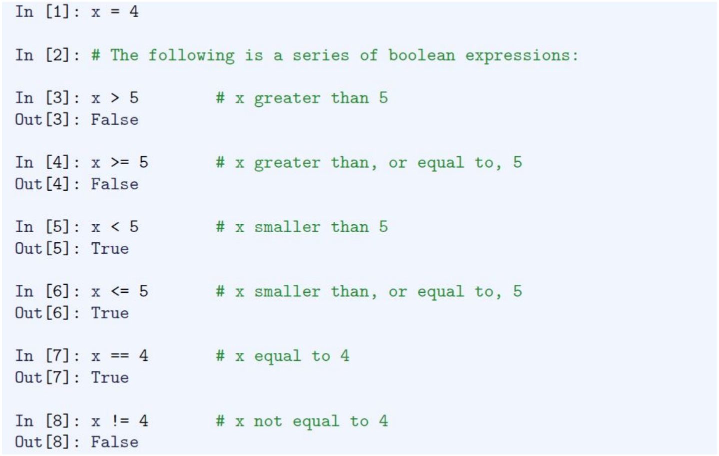
- for loop runs for a specified number of iterations
- The other basic loop construction in Python is the while loop, which runs as long as a condition is `True`

Boolean expressions

- aka *logical expressions*
- these evaluate to *Boolean values* `True` and `False`
  - note capital letters T and F
- there are 6 *relational operators* in Python—comparing values `>`, `<`, `>=`, `<=`, `==` and `!=`
- typeset as simple table

Relational operators: comparing values

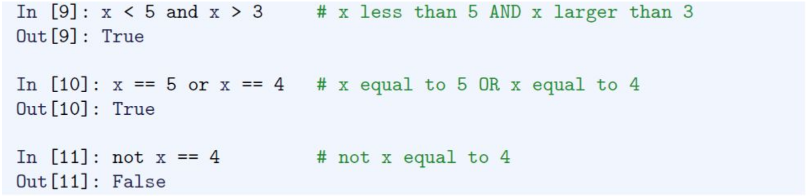
Live demo of relational operators



```
In [1]: x = 4

In [2]: # The following is a series of boolean expressions:

In [3]: x > 5        # x greater than 5
Out[3]: False

In [4]: x >= 5       # x greater than, or equal to, 5
Out[4]: False

In [5]: x < 5        # x smaller than 5
Out[5]: True

In [6]: x <= 5       # x smaller than, or equal to, 5
Out[6]: True

In [7]: x == 4       # x equal to 4
Out[7]: True

In [8]: x != 4       # x not equal to 4
Out[8]: False
```

Boolean operators: `and`, `or`, `not`

Live demo of Boolean operators



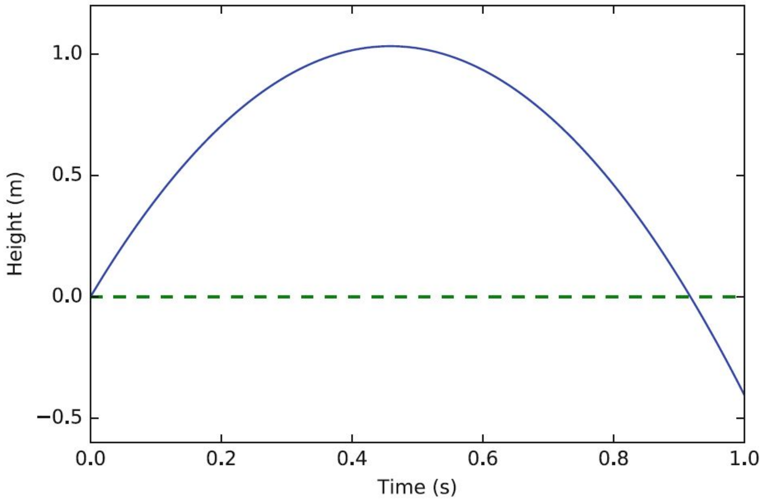
```
In [9]: x < 5 and x > 3      # x less than 5 AND x larger than 3
Out[9]: True

In [10]: x == 5 or x == 4    # x equal to 5 OR x equal to 4
Out[10]: True

In [11]: not x == 4          # not x equal to 4
Out[11]: False
```

- Boolean variable type
  - int, float, str, bool
- Boolean values may be combined into longer expressions by use of `and` and `or`
- basics of Boolean operators: week 1 Thurs lecture
  - covered in *much* more depth in ELEC1710

Example: Finding the Time of Flight

- context/description
- we modify/extend earlier example

Ball height vs. time

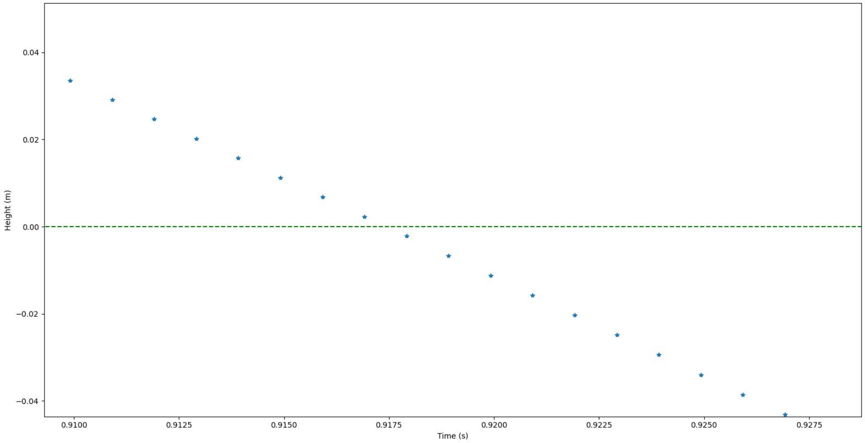


```python
import numpy as np

v0 = 4.5                      # 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

# Find index where ball approximately has reached y=0
i = 0
while y[i] >= 0:
    i = i + 1

# Since y[i] is the height at time t[i], we do know the
# time as well when we have the index i...
print('Time of flight (in seconds): {:g}'.format(t[i]))

# We plot the path again just for comparison
import matplotlib.pyplot as plt
plt.plot(t, y)
plt.plot(t, 0*t, 'g--')
plt.xlabel('Time (s)')
plt.ylabel('Height (m)')
plt.show()
```

Python code: `ball_time.py`

```
# Find index where ball approximately has reached y=0
i = 0
while y[i] >= 0:
    i = i + 1
```

- slowly and meticulously consider y[i] >= condition
- index i after loop is index to first element of array that is negative
- confirm in Python



- plot as * then zoom in to see time where crossing occurs

Structure of a typical `while` loop

```
while some_condition:   # Loop header
    <code line 1>           # 1st line in loop body
    <code line 2>           # 2nd line in loop body
    ...
    ...
# This is the first line after the loop
```

- first line is *while loop header*
  - reserved word `while`, ends with colon, both necessary
- indented lines after header are a *block* of statements
  - called the *loop body*
- indentation is 4 spaces by convention
- once indentation is reversed, loop body has ended

- `some_condition` is a Boolean expression
  - must evaluate to `True` or `False`

- if `some_condition` is initially `False`:
  - loop body statements are *never* executed

- if `some_condition` is initially `True`:
  - statements in loop body are evaluated once
  - `some_condition` evaluated again
  - ...and the process continues

Summary: `while` loop runs until the Boolean expression `some_condition` becomes `False`

Infinite Loops

- It is possible to have a while loop in which the condition never evaluates to False, meaning that program execution can not escape the loop!
- this is referred to as an infinite loop

3) branching: if, elif and else)

- context
- extended Example: Judging the Water Temperature
- will build up a program in stages

One if-test

- screenshot/code LLp68

Two if-tests

- LLp68

An if-else Construction

- LLp69

An if-elif-else Construction

- LLp69

general form of an if-elif-else

- §3.3.2

branching summary

- if
- if / else
- if / elif / else

Lecture summary