

ENGG1003 - Monday Week 3

Loops and branching

Steve Weller

University of Newcastle

8 March, 2021

Last compiled: March 7, 2021 4:03pm +11:00

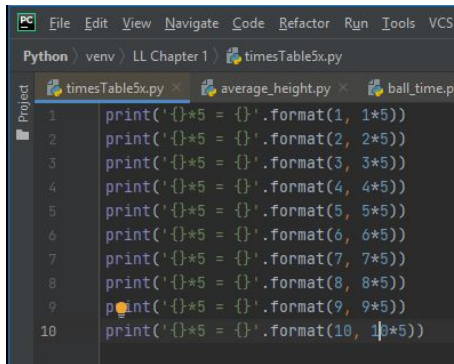
Lecture overview

- 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

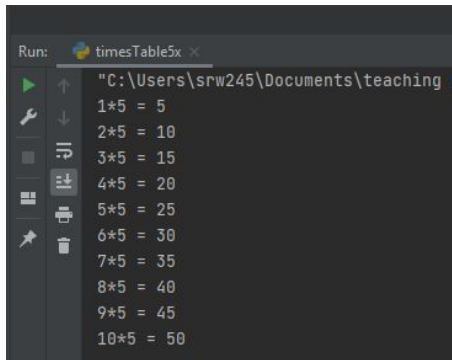
1) Iteration using `for` loop

- many computations for solving engineering problems are intrinsically repetitive
- all programming languages have certain *loop* structures to enable repetitive code execution
- Python provides two such structures:
 - ▶ `for` loop
 - ▶ `while` loop
- **Motivation:** print the 5 times table at the console

Live demo: 5 times table, brute force



```
File Edit View Navigate Code Refactor Run Tools VCS
Python > venv > LL Chapter 1 > timesTable5x.py
timesTable5x.py average_height.py ball_time.p
1 print('{}*5 = {}'.format(1, 1*5))
2 print('{}*5 = {}'.format(2, 2*5))
3 print('{}*5 = {}'.format(3, 3*5))
4 print('{}*5 = {}'.format(4, 4*5))
5 print('{}*5 = {}'.format(5, 5*5))
6 print('{}*5 = {}'.format(6, 6*5))
7 print('{}*5 = {}'.format(7, 7*5))
8 print('{}*5 = {}'.format(8, 8*5))
9 print('{}*5 = {}'.format(9, 9*5))
10 print('{}*5 = {}'.format(10, 10*5))
```



```
Run: timesTable5x x
"C:\Users\srw245\Documents\teaching
1*5 = 5
2*5 = 10
3*5 = 15
4*5 = 20
5*5 = 25
6*5 = 30
7*5 = 35
8*5 = 40
9*5 = 45
10*5 = 50
```

Our first loop

- using `for` loop, can replace 10 lines of code with just 2 lines:

```
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
```

- *loop variable* `i` takes on each of the values 1 to 10
- ...and for each value the `print` function is called

A typical `for` loop

```
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
```

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

Nested loops

- for each iteration of a loop... execute *another* loop!
- one loop “inside” another, hence *nested*
- *two* levels of indentation are needed

```
for i in [1, 2, 3]:  
    # First indentation level (4 spaces)  
    print('i = {}'.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

- elements of each array are identified by an index
- for loop can use array index as a loop variable

Example: compute average of five numbers

$$\frac{h[0] + h[1] + h[2] + h[3] + h[4]}{5}$$

```
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: {0:g} meter'.format(average))
```


Live demo: `for` loop

- Python code: `average_height.py`

Using the range function

- what about `for` loop with *really large* number of iterations?

✗ `for i in [0, 1, 2, 3, 4 ..., 9999]:`

- built-in function `range` solves this problem
- instead of `for i in [0, 1, 2, 3, 4]:`
- ...use this `for i in range(0, 5, 1):`
- general form of call to `range` as follows

```
for loop_variable in range(start, stop, step):
```

- ▶ ...start at integer `start`
 ...increment by integer `step`
 ...stop before integer `stop`

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

Comparing values

Live demo

```
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

```
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

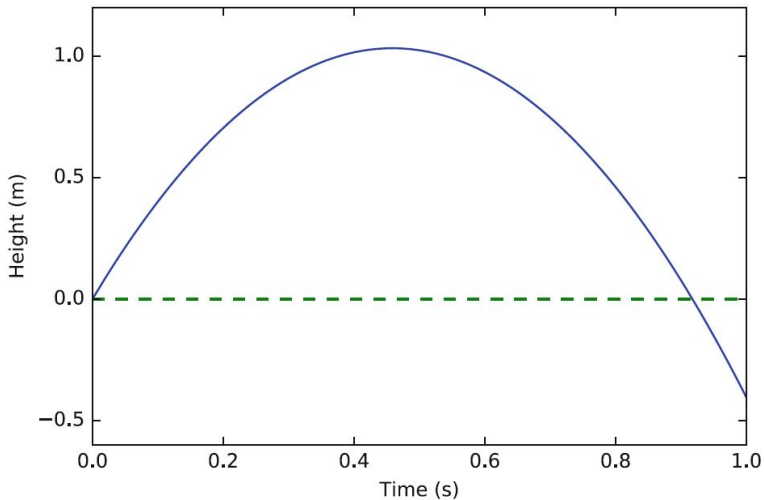
In [11]: not x == 4          # not x equal to 4
Out[11]: False
```

- 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
- now initial velocity is only 4.5 m/s
 - ▶ was 5 m/s in last weeks lecture
- calculate height for 1 s and find “height” is negative for sufficiently large t —this is deliberate

Ball height vs. time




```

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]))

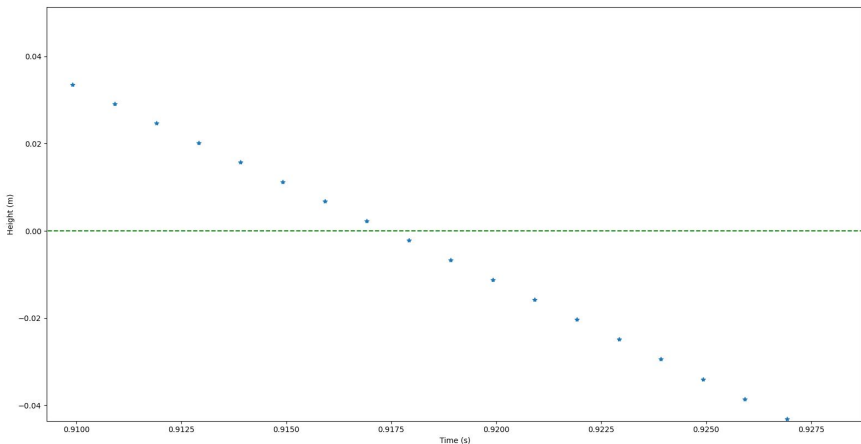
# We plot the path again just for comparison
import matplotlib.pyplot as plt
plt.plot(t, y)
plt.plot(t, 0*t, 'g--')
plt.xlabel('Time (s)')
plt.ylabel('Height (m)')
plt.show()

```

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 $y[i] \geq 0$ 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

- Possible to have a `while` loop in which the condition *never* evaluates to `False`
 - ▶ program execution cannot escape the loop!
- Referred to as an *infinite loop*
- Might be deliberate ...
 - ▶ program runs “forever”, eg: surveillance camera system
- ... but infinite loops are usually unintentional
 - ▶ result of a program *bug*
 - ▶ `Ctrl+c` to stop program

3) Branching: `if`, `elif` and `else`

Assume we want to write a program that helps us decide, based on water temperature alone (in degrees Celcius), whether we should go swimming or not.

- context
- extended Example: Judging the Water Temperature
- will build up a program in stages
 - ▶ programming as a step-wise process

One if-test

```
T = float(input('What is the water temperature? '))
if T > 24:
    print('Great, jump in!')
# First line after if part
```

- XXX

Two if-tests

```
T = float(input('What is the water temperature? '))
if T > 24:                                     # testing condition 1
    print('Great, jump in!')
if T <= 24:                                    # testing condition 2
    print('Do not swim. Too cold!')
# First line after if-if construction
```

● XXX

An if-else construction

```
T = float(input('What is the water temperature? '))
if T > 24:                                # testing condition 1
    print('Great, jump in!')
else:
    print('Do not swim. Too cold!')
# First line after if-else construction
```

● XXX

An if-elif-else construction

```
T = float(input('What is the water temperature? '))
if T > 24:                                     # testing condition 1
    print('Great, jump in!')
elif 20 <= T <= 24:                             # testing condition 2
    print('Not bad. Put your toe in first!')
else:
    print('Do not swim. Too cold!')
# First line after if-elif-else construction
```

● XXX

General form of an if-elif-else

```
if condition_1:                # testing condition 1
    <code line 1>
    <code line 2>
    ...
elif condition_2:              # testing condition 2
    <code line 1>
    <code line 2>
    ...
elif condition_3:              # testing condition 3
    <code line 1>
    <code line 2>
    ...
else:
    <code line 1>
    <code line 2>
    ...
# First line after if-elif-else construction
```

Lecture summary

- Iteration using **for** loop
 - ▶ fixed number of iterations
- Iteration using **while**
 - ▶ keep iterating whenever a Boolean condition is satisfied
- Branching: **if**, **elif** and **else**
 - ▶ conditional execution of code blocks
 - `if`
 - `if-else`
 - `if-elif-else`