COMP90059 Semester 1, 2022
Introduction to Programming

THE UNIVERSITY OF
MELBOURNE

Lecture 6: Iteration

Dr Simon D'Alfonso

School of Computing and Information Systems
Faculty of Engineering and Information Technology

Lecture 5 Challenges

1. Write a function ascii_match(), which accepts two required input arguments, an integer and a string. If the ASCII numbers of all the characters in the string add up to the value of the input integer, return True. Otherwise, return False.
2. Write a function avg_three() that returns the average (mean) of three integers between 1 and 9. The function should allow for three optional input arguments (num1, num2, num3), but if any of these arguments is not provided when the function is called, then a random number is generated for that value. See https://www.w3schools.com/python/ref_random_randint.asp for how to generate random integers.

Notes

An update on the current plans for the Mid-Semester Test (MST) and Assignment 1 (A1):

- A1 will be released via Grok on the Friday at the end of Week 6 (April 8) and will be due 2 weeks after that.
- The MST will be held during the lecture period of Week 7. It will be conducted online via LMS. I will put up some sample questions soon.
- With MST, be careful in cut and pasting.
- Don’t worry.
- Discussion forum: tutors will also respond.

Overview

- For Loops
- While loops
- Nested loops
- Good practice for writing code and the PEP8 standard

Iteration

- Repeating or looping instructions to make the computer do something repeatedly.
- Repeat something a fixed number of times:
  - times a number by itself *n* times
  - print each element in a collection
  - print 7 instances of something
- Repeat something until something happens (e.g., scroll while button held)

Loops in Python

- The Python programming language provides the following types of loops to handle looping requirements:
  - *for* loop - iterates over the members of a sequence in order, executing the statement(s) inside each time.
  - *while* loop - repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body.
- These loop types can also be nested: can use one or more loops inside any another

The *for* loop

- A *for* loop is used for iterating over Python data collections (strings, lists, tuples, dictionaries, sets)
- During each pass or iteration of the loop statement(s) inside will be executed
- Basic form:

```
for <interating_var> in <collection>:
    statement(s)
```

- Note: *in* here is not (quite) the same as the comparison operator of the same name

For loops and sequences

- So for loops are used with sequences, which are ordered collections
- We have seen three types of sequences so far (to varying degrees of detail):
  - Strings
  - Lists
  - Tuples

For Loop and String Example

```
for char in 'COMP90059':
    print(char)
```

For Loop and Tuple Example

```
sum = 0
for i in (1, 2, 3):
    sum = sum + i
print(sum)
```

The code above is equivalent to the code below.

```
sum = 0
sum = sum + 1
sum = sum + 2
sum = sum + 3
print(sum)
```

For Loop and List Example

```
sum = 0
for i in [1, 2, 3]:
    sum = sum + i
print(sum)
```

The code above is equivalent to the code in the previous slide.

Exercise 1

Write a function is_pangram(s), which determines if an input string *s* is a pangram and returns True or False accordingly. Recall that a pangram is a string in which each alphabetical characters appears at least once.

Exercise 1 Solution

```
def is_pangram(s):
    something = ? #what goes here
    for char in something:
        if char not in s.lower():
            return False

    return True
```

Using range()

- The useful Python function range() returns a sequence of numbers.
- It is used with loops such as *for* to conveniently iterate a certain number of times.
- Take the two code examples below to print out the numbers 1, 2, 3.

```
#approach using a tuple with ordered numbers each specified.
for i in (1, 2, 3):
    print(i)

#approach using a range
for i in range(1, 4):
    print(i)
```

- If we needed to go through 1 – 1000, the latter approach would be much better than the former.

Range()

There are three ways you can call range():

- range(stop) takes one argument.
  - Starts at 0 and stops at but does not include *stop*
  - For example, range(3) would generate the numbers 0, 1, 2
- range(start, stop) takes two arguments.
  - Starts at integer *start* and *stop* is same as above.
- range(start, stop, step) takes three arguments.
  - *Start* and *stop* same as above, *step* argument determines the increment between each integer in the sequence

Range() Examples

```
for i in range(3):
    print(i)

for i in range(5, 10):
    print(i)

for i in range(0, 10, 2):
    print(i)
```

Points to remember about range()

- All three arguments must be integers (positive or negative).
- The step value must not be zero. If a step is zero python raises a ValueError exception.
- range() is a type in Python
- Users can access items in a range() by index, just as can be done with strings, lists and tuples.
- Technically *start* can be greater than *stop*, though this would effectively give a null range.

Range() Examples

```
for i in range(1, 10, 0):
    print(i)

for i in range(-10, -1):
    print(i)

for i in range(10, 5):
    print(i)
```

For over lists

- Lists (arrays) are sequences that are mutable
- They can contain various types of data, including mixtures of different data types
- Share many similarities with strings, and can similarly be accessed with indexes

```
>>> highest_scorers = ['Sally', 'Jessica', 'Anne']
>>> highest_scorers[0]
'Sally'
>>> highest_scorers[2] = 'Brenda'
>>> highest_scorers[-1]
'Brenda'
```

List Methods

- Python has a set of built-in methods that you can use for lists: https://www.w3schools.com/python/python_lists_methods.asp

```
>>> my_numbers = [4, 5, 6.6, 3.3]
>>> my_numbers.append(1)
>>> my_numbers.sort()
>>> my_numbers
[1, 3.3, 4, 5, 6.6]

>>> my_numbers = [4, 5, 6.6, 3.3, 1]
>>> sorted(my_numbers) #sorted sorts collections
[1, 3.3, 4, 5, 6.6]
>>> my_numbers
[4, 5, 6.6, 3.3, 1]
```

List Sort Trivia

What would happen with the following:

```
>>> my_list = [5, 6, 1, False]
>>> sorted(my_list)
?


>>> my_list = [4, 'four', True,
3.3]
>>> sorted(my_list)
?
```

For, range() and lists

```
# using range for iteration over a list
my_list = [10, 20, 30, 40]
for i in range(len(my_list)):
    print(my_list[i], end=" ")
print()
```

The *while* loop

- The other type of loop is a *while* loop, which is a conditional loop.
- The general idea is that we continue repeating a block of code as long as a given condition holds.
- Basic form:

```
while <condition(s)>:
    statement(s)
```

While Loop Example

```
number_guess = None
while number_guess != "7":
    number_guess = input('Guess the lucky number: ')

print(number_guess + " is correct!")
```

Break statement

Another way to end loops is via a break in the block of code. This prematurely and unconditionally exits from the loop. Below is an example that is an alternative to the code in the previous slide.

```
number_guess = None
while True:
    number_guess = input('Guess the lucky 
number: ')
    if number_guess == "7":
        break
    print('Sorry, wrong guess')

print(number_guess + " is correct!")
```

Break example with *for* loop

```
my_list = [1, 5, 8, 9]
for i in my_list:
    print(i)
    if i == 8:
        break
```

What is the output of this program?

Beware the infinite loop

```
while True:
    print(True):

#---

i = 1
j = 10
while i < j:
    print(i)
```

How could we do something so that we keep print(i) but it stops after printing out 1, 2, 3, …

Choosing between *for* and *while* loops

- If you need to iterate over all items of an iterable, use a *for* loop
- If there is a well defined end-point to the iteration which doesn’t involve iterating over all items, use a *while* loop
- With a *for* loop, avoid modifying the object you are iterating over within the block of code
- Given a choice between the two, *for* loops are generally more elegant/safer/easier to understand

Exercise 2

- The Fibonacci Sequence is the series of numbers: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...
- Starting with 0 and 1, the next number is found by adding up the two numbers before it.
- Write a function that gets an integer *n* via input() and using *for* generates a list with the first *n* Fibonacci numbers.

Exercise 2 Solution

```
n = int(input("Number of elements in Fibonacci Series, n, (n>=2) :
"))

#initialize the list with starting elements: 0, 1
fibonacci_list = [0, 1]

if n > 2:
    for i in range(2, n):
        #next elment in series = sum of its previous two numbers
        next_element = fibonacci_list[i-1] + fibonacci_list[i-2]
        #append the element to the series
        fibonacci_list.append(next_element)

print(fibonacci_list)
```

Nested *for* loops

Syntax for nested *for*:

```
for var1 in sequence:
    for var2 in sequence:
        statements(s) # inner loop
    statements(s) # outer loop

statement(s) # outside loops
```

Nested *for* Example 1

```
n = int(input('enter a number:'))

for i in range(1, n+1):
    for j in range(1, n+1):
        print(i, j)

print("good bye")
```

Nested *for* Example 2

```
#Example code that checks whether
numbers between 1 and 10 are prime.

for num in range(1, 11):
    prime = True
    for x in range(2, num):
        if not(num % x):
            prime = False
    if prime:
        print(num, "is a prime")
    else:
        print(num, "is not a prime")
```

Nested *for* Example 3

```
#display the 12x12 multiplication tables

for i in range(1, 13):
    for j in range(1, 13):
        print(i * j, ' ' * (3 - len(str(i * j))), '|', end = "")
    print('')
```

Nested *while*

Syntax for nested while:

```
while <condition(s)>:
    while <condition(s)>:
        statement(s) # inner loop
    statement(s) # outer loop

statement(s) # outside loops
```

Using *for/while* and *else*

In Python we can use *else* with a *for/while* loop. The else block will be executed only if the loop terminates naturally (not due to a break).

```
def digits_in_word(word):
    for i in range(len(word)):
        if word[i].isdigit():
            print("The word contains digits")
            break
    else:
        print("The word does not contain digits")
```

Good practice for writing code

- PEP8 provides a set of standards or best practices for writing good, neat code: https://peps.python.org/pep-0008/

- Why a standard:
  - To improve readability; code is read more often than it is written.
  - Make code more consistent, which promotes code sharing and reusability.

Overview of PEP8

- 4 spaces per indent line level.
- Lines must not exceed 79 characters.
- Use blank lines to separate logical sections (e.g., between functions and major sections of code associated with a given comment).
- Always start a new line after *if*, *elif*, *else*, *while*, *for*, etc.
- Function names should be lowercase, with words separated by underscores.
- Constants should be written in all capitals , with words separated by underscores.
- Don’t compare Boolean values to True or False using ==

Summary

Today we covered:

- The for loop
- The range function
- The while loop
- Nested loops
- Good coding style

Lecture 6 Challenges

- Convert the code on the Nested for Example 1 slide to a functionally equivalent piece of code that uses *while* loops instead of *for* loops.
- Modify the Exercise 1 Solution so that the function instead returns a list containing all characters that don't appear in the string. If the string is a pangram, then return an empty list.
- Write a program that uses a while loop to print out each individual character in a given string line by line.

Lecture Identification and Acknowledgement

Coordinator / Lecturer: Simon D'Alfonso

Semester: Semester 1, 2022

These slides include materials from 2020 - 2021 instances of COMP90059 run by Kylie McColl or Wally Smith