Digital World (2018)

Week 4, S1: For-Loops and Nested Lists

Chris Poskitt

SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN

Refresher Question 1
b.socrative.com, POSKITT5665

What is printed by the following code? (Output is on one line to save space.)

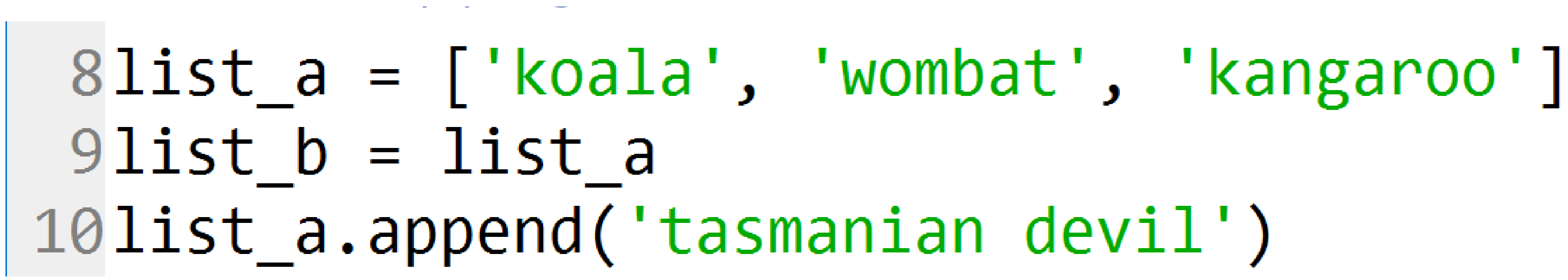
```
x = 6
while x > 4:
  x = x - 1
  print(x)
```

- A. 6 5
- B. 6 5 4
- C. 5 4
- D. 5 4 3
- E. 6 5 4 3

Acknowledgements: Prof. Norman

Refresher Question 2

b.socrative.com, POSKITT5665



```python
list_a = ['koala', 'wombat', 'kangaroo']
list_b = list_a
list_a.append('tasmanian devil')
```

len(list_a) == len(list_b) gives

A. True B. False

Acknowledgements: Prof. Norman

for-loops

```
def mystery_function(list):
    x = 0
    for element in list:
        x = x + element
    return x
```

list = [1, 3, 5]

for-loops

```
def mystery_function(list):
    x = 0
    for element in list:
        x = x + element
    return x
```

list = [1, 3, 5]

for-loops

```
def mystery_function(list):
    x = 0
    for element in list:
        x = x + element
    return x
```

list = [1, 3, 5]

x ——> 0

for-loops

```
  def mystery_function(list):
      x = 0
      for element in list:
→         x = x + element
      return x
```

list = [$\underset{\overset{\uparrow}{\text{element}}}{1}$, 3, 5]

x ——> 0

for-loops

```
def mystery_function(list):
    x = 0
    for element in list:
        x = x + element
    return x
```

list = [1, 3, 5]

$$\begin{tikzcd} 1 \\ \text{element} \arrow[u] \end{tikzcd}$$

x ——> 1

for-loops

```
def mystery_function(list):
    x = 0
    for element in list:
        x = x + element
    return x
```

list = [1, 3, 5]

element

x ——> 1

for-loops

```
def mystery_function(list):
    x = 0
    for element in list:
        x = x + element
    return x
```

list = [1, 3, 5]

element

x ——> 4

for-loops

```
def mystery_function(list):
    x = 0
    for element in list:
➡       x = x + element
    return x
```

list = [1, 3, 5]

$$\uparrow$$

element

x ———> 4

for-loops

```
def mystery_function(list):
    x = 0
    for element in list:
        x = x + element
    return x
```

list = [1, 3, 5]

element

x ——> 9

for-loops

```
def mystery_function(list):
    x = 0
    for element in list:
        x = x + element
    return x
```

list = [1, 3, 5]

$\uparrow$

element

$x \longrightarrow 9$

for-loops: an equivalent function

```
def mystery_function(list):
    x = 0
    for index in range(len(list)):
        x = x + list[index]
    return x
```

for-loops: an equivalent function

```
def mystery_function(list):
    x = 0
    for index in range(len(list)):
        x = x + list[index]
    return x
```

what does the ***range*** *function do?*

range function

- produces an immutable sequence of numbers

```
range(stop)
```

```
range(start, stop)
```

```
range(start, stop, step)
```

range function

- produces an immutable sequence of numbers

range(stop)

range(start, stop)

range(start, stop, step)

***range** is "lazily" executed — returns numbers only when needed*
*…it does *not* create a list!*

for-loops: an equivalent function

```
def mystery_function(list):
    x = 0
    for index in range(len(list)):
        x = x + list[index]
    return x
```

list = [1, 3, 5]

for-loops: an equivalent function

```
def mystery_function(list):
    x = 0
    for index in range(len(list)):
        x = x + list[index]
    return x
```

list = [1, 3, 5]

for-loops: an equivalent function

```
def mystery_function(list):
    x = 0
    for index in range(len(list)):
        x = x + list[index]
    return x
```

list = [1, 3, 5]

x ——> 0

for-loops: an equivalent function

range(len([1, 3, 5]))
= 0, …

$\uparrow$

index

```
def mystery_function(list):
    x = 0
    for index in range(len(list)):
        x = x + list[index]
    return x
```

list = [1, 3, 5]

x ——> 0

for-loops: an equivalent function

```
def mystery_function(list):
    x = 0
    for index in range(len(list)):
        x = x + list[index]
    return x
```

range(len([1, 3, 5]))
= 0, …

↑

index

list = [1, 3, 5]

↑

list[index]

x ——> 0

for-loops: an equivalent function

```
def mystery_function(list):
    x = 0
    for index in range(len(list)):
        x = x + list[index]
    return x
```

range(len([1, 3, 5]))
= 0, ...

$\uparrow$

index

list = [1, 3, 5]

$\uparrow$

list[index]

x ——> 1

for-loops: an equivalent function

```
def mystery_function(list):
    x = 0
    for index in range(len(list)):
        x = x + list[index]
    return x
```

range(len([1, 3, 5]))
= 0, 1, …

↑

index

list = [1, 3, 5]

↑

list[index]

x ——> 1

for-loops: an equivalent function

```
def mystery_function(list):
    x = 0
    for index in range(len(list)):
        x = x + list[index]
    return x
```

x ——> 4

range(len([1, 3, 5]))
= 0, 1, ...

↑
index

list = [1, 3, 5]

↑
list[index]

for-loops: an equivalent function

$$\texttt{range(len([1, 3, 5]))} = 0, 1, 2$$
$$\uparrow$$
index

```
def mystery_function(list):
    x = 0
    for index in range(len(list)):
        x = x + list[index]
    return x
```

list = [1, 3, 5]

$$\uparrow$$
list[index]

x ——> 4

for-loops: an equivalent function

range(len([1, 3, 5]))
= 0, 1, 2

↑

index

```
def mystery_function(list):
    x = 0
    for index in range(len(list)):
        x = x + list[index]
    return x
```

list = [1, 3, 5]

↑

list[index]

x ———> 9

for-loops: an equivalent function

```
def mystery_function(list):
    x = 0
    for index in range(len(list)):
        x = x + list[index]
    return x
```

range(len([1, 3, 5]))
= 0, 1, 2

↑

index

list = [1, 3, 5]

↑

list[index]

x ——> 9

while vs. for

- when is a while-loop better?
- when is a for-loop better?

2. *Functions: Compound value:* Suppose you deposit \$100 on the first day of each month into a savings account with an annual interest rate of 5%. The bank calculates the interest gained and credits the amount to you at the end of the month. The monthly interest rate is 0.05/12=0.00417. At the end of the first month, the value in the account is

   $$100 * (1 + 0.00417) = 100.417$$

   At the end of the second month, the value in the account is

   $$(100 + 100.417) * (1 + 0.00417) = 201.252$$

   At the end of the third month, the value in the account is

   $$(100 + 201.252) * (1 + 0.00417) = 302.507$$

   and so on.

   Write a function named `compound_value_months` that takes in a monthly saving amount, an annual interest rate, and the number of months (n), and returns the account value at the end of the n^{th} month. Round the return value to 2 decimal places. Note: this problem is similar to one of the problems you did in the past. The only different is that the number of months here can be any integer n, thus requiring you to use loops.

```
>>> ans=compound_value_months(100,0.05,6)
>>> print(ans)
608.81
```

Refresher: Sublists

list[start:stop]

list[start:stop:step]

Refresher: Sublists

list[start:stop]

list[start:stop:step]

stop but don't include!

Refresher: Sublists

list[start:stop]

list[start:stop:step]

stop but <u>don’t</u> include!

list = [‘a’, ‘b’, ‘c’, ‘d’, ‘e’, ‘f’, ‘g’]

list[:] list[2:5] list[-5:-2] list[::2] list[::-1]

Refresher: Sublists

$$\boxed{\texttt{list[start:stop]}} \qquad \boxed{\texttt{list[start:stop:step]}}$$

stop but <u>don't</u> include!

```
         -7  -6  -5  -4  -3  -2  -1
list = ['a', 'b', 'c', 'd', 'e', 'f', 'g']
          0   1   2   3   4   5   6
```

list[:]   list[2:5]   list[-5:-2]   list[::2]   list[::-1]

Nested Lists: *lists can contain lists!*

- e.g. a matrix can be represented as a nested list

$$M = \begin{bmatrix} 0 & 0 & 2 & 1 \\ 5 & 5 & 3 & 1 \\ 33 & 66 & 77 & 99 \end{bmatrix}$$

Nested Lists: *lists can contain lists!*

- e.g. a matrix can be represented as a nested list

$$M = \begin{bmatrix} 0 & 0 & 2 & 1 \\ 5 & 5 & 3 & 1 \\ 33 & 66 & 77 & 99 \end{bmatrix}$$

M = [[0, 0, 2, 1], [5, 5, 3, 1], [33, 66, 77, 99]]

*- how do we **access** M's elements?*
*- how do we **iterate** across M?*

Nested Lists: *lists can contain lists!*

- e.g. a matrix can be represented as a nested list

$$M = \begin{bmatrix} 0 & 0 & 2 & 1 \\ 5 & 5 & 3 & 1 \\ 33 & 66 & 77 & 99 \end{bmatrix}$$

M = [[0, 0, 2, 1], [5, 5, 3, 1], [33, 66, 77, 99]]

M[0] M[1] M[2]

- *how do we **access** M's elements?*
- *how do we **iterate** across M?*

Nested Lists: *lists can contain lists!*

- e.g. a matrix can be represented as a nested list

$$M = \begin{bmatrix} 0 & 0 & 2 & 1 \\ 5 & 5 & 3 & 1 \\ 33 & 66 & 77 & 99 \end{bmatrix}$$

M[0][0]
M[0][1]
… etc.

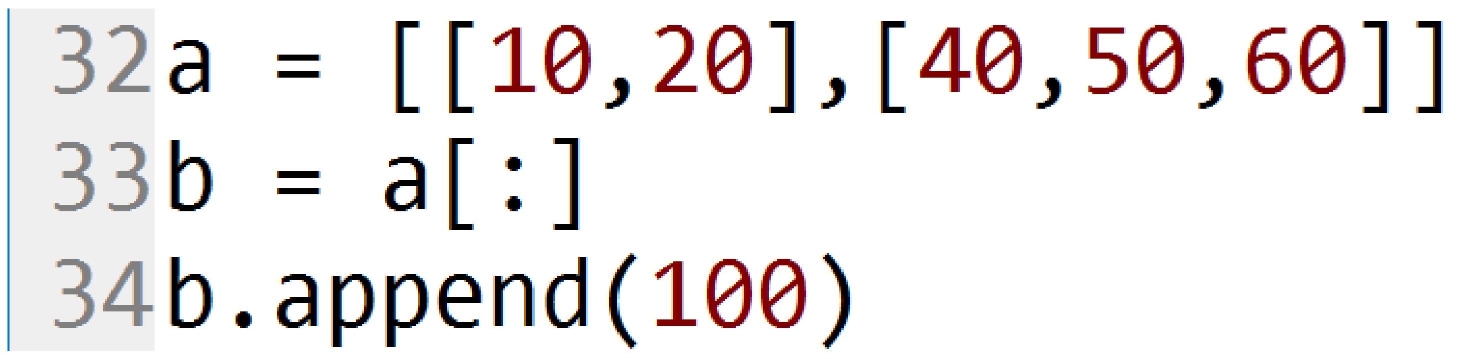
M = [[0, 0, 2, 1], [5, 5, 3, 1], [33, 66, 77, 99]]

M[0] M[1] M[2]

- *how do we **access** M's elements?*
- *how do we **iterate** across M?*

Clicker Question

b.socrative.com, POSKITT5665



```
a = [[10,20],[40,50,60]]
b = a[:]
b.append(100)
```

`a[2] == 100` evaluates to True.   A. True   B. False

Acknowledgements: Prof. Norman

3. *Loops:* Write a function named **`find_average`** that takes in a list of lists as an input. Each sublist contains numbers. The function returns a list of the averages of each sublist, and the overall average. If the sublist is empty, take the average to 0.0.

   For example, if the input list is $[[3,4],[5,6,7],[-1,2,3]]$, the program returns the list $[3.5, 6.0, 1.333]$, and the overall average 3.625, calculated by summing all the numbers in all the sublists and dividing this total sum by the total count of all the numbers.

```
>>> ans=find_average([[3,4],[5,6,7],[-1,2,8]])
>>> print(ans)
([3.5, 6.0, 3.0], 4.25)

>>> ans=find_average([[13.13,1.1,1.1],[],[1,1,0.67]])
>>> print(ans)
```

Summary

- for-loops vs. while-loops
- iterating over sequences using range
- obtaining sublists through slicing
- iterating nested lists using nested for-loops
- *for next time:* try to complete questions CS1 and CS4