Announcements

- Tutorial room changes
  - BYOD lab Weds 6-8pm
    - Was in Marie Reay 5.02
    - Now in Birch 1.35/1.36 (at the same time a before)
  - BYOD labs in Hanna Neumann Bldg 1.25
    - The screen is not going to be fixed
    - Labs are now moved to: TBA
      - Thur 3-5pm - TBA
      - Fri 12-2pm - TBA
      - Fri 2-4pm - TBA
- Homework 2 due on Sun 11:55pm
- Quiz 2 and 3

Recursion

COMP1730/COMP6730

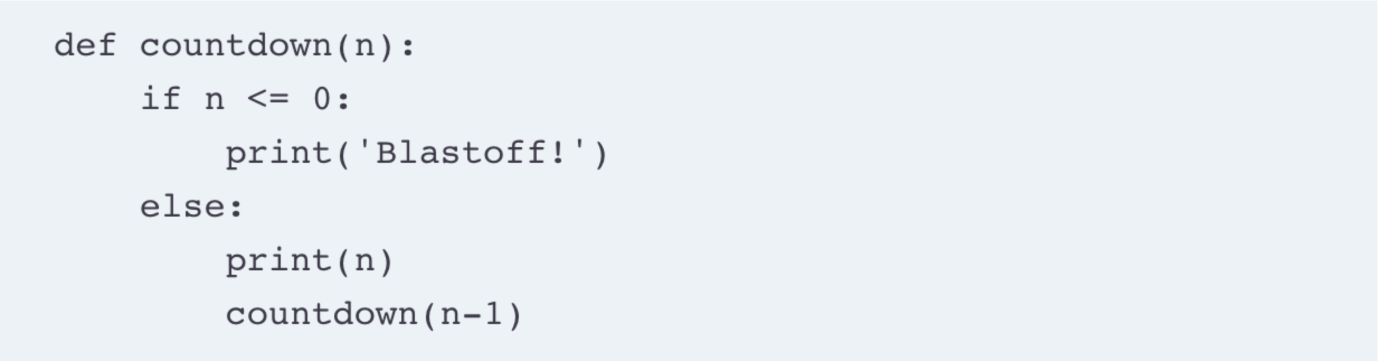
Reading: Textbook chapter 5 : Alex Downey, *Think Python*, 2nd Edition (2016) from '*Recursion*' section to end of chapter

Sections: Ch 5 – *Recursion, Stack Diagrams for Recursive Functions, Infinite Recursion*

Australian
National
University

Recursion

- Definition: use of a procedure, subroutine or function that calls itself one or more times until a specified condition is met
- In Python – and other languages – a function can call itself:



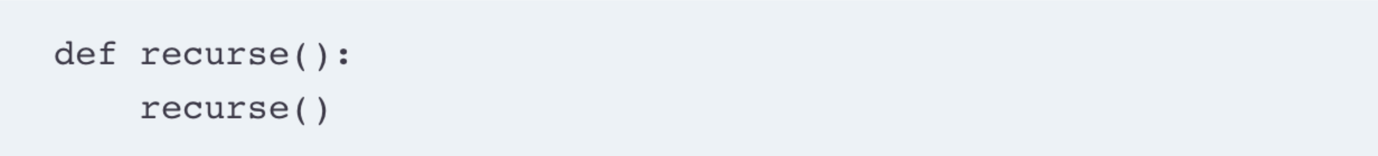
```
def countdown(n):
    if n <= 0:
        print('Blastoff!')
    else:
        print(n)
        countdown(n-1)
```

Downey (2015) Think Python, 2nd Ed.

- Why would you want this? It is very useful.
- It is a way to repeat an operation easily, with altered input
- Recursion is a way to think about solving a problem: how to reduce it to a simpler instance of itself?

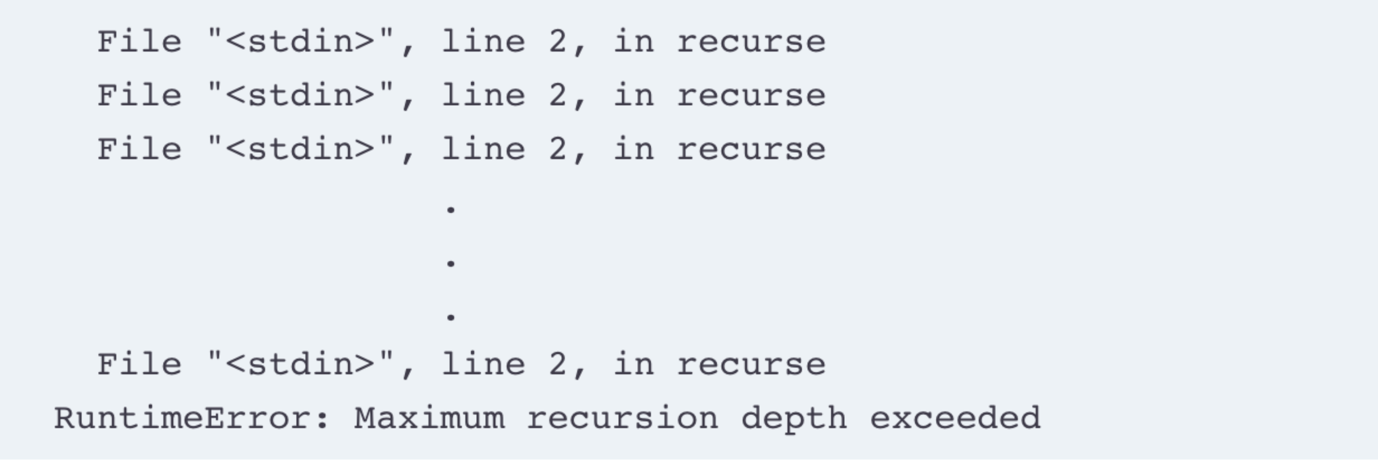
Infinite recursion (the curse of)

- Recursion requires a conditional, branching statement, so that it does not recurse for ever. So, not like this:



```
def recurse():
    recurse()
```

- Infinite recursion is a common error that we will all encounter. In python, infinite recursion is *automagically* terminated, to save us from ourselves:



```
  File "<stdin>", line 2, in recurse
  File "<stdin>", line 2, in recurse
  File "<stdin>", line 2, in recurse
                  .
                  .
                  .
  File "<stdin>", line 2, in recurse
RuntimeError: Maximum recursion depth exceeded
```

Downey (2015) Think Python, 2nd Ed.

The call stack (reminder)

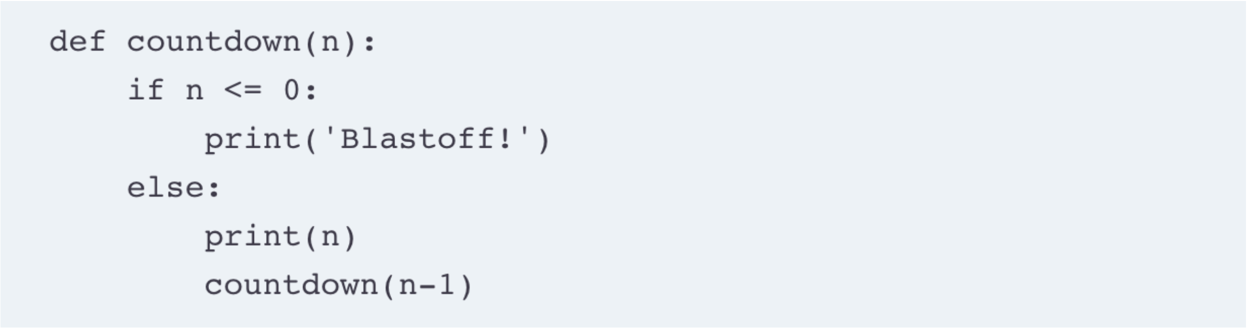
- When a function call begins, the current instruction of the caller function is put “on a stack”
- The called function ends when it encounters a `return` statement, or reaches the end of the block
- The interpreter then returns to the next instruction after where the function was called
- The *call stack* keeps track of where to come back to after each current function call

Example

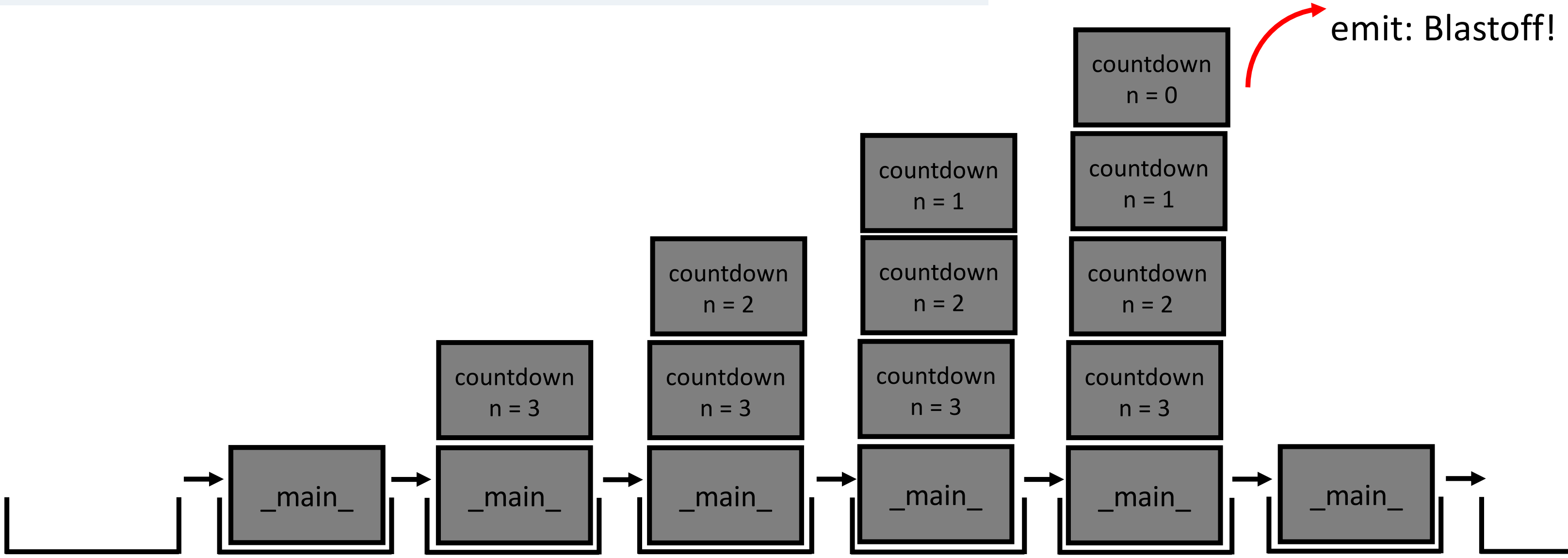
- Recursion – blast-off example

The call stack with recursion

Downey (2015) Think Python, 2nd Ed.



```
def countdown(n):
    if n <= 0:
        print('Blastoff!')
    else:
        print(n)
        countdown(n-1)
```



Exercises

- Complete Exercises 5-4 and 5-5 of *Think Python*.

Reading

- Chapter 5 of *Think Python*
  - Sections: Ch 5 – *Recursion, Stack Diagrams for Recursive Functions, Infinite Recursion*

Iteration

COMP1730/3730

Reading:

Think Python, 2nd Edition (2016), Ch 7 sections *‘the while statement*’ and ’*break*’ AND Chapter 4: Simple Repetition (`for` loops)

Intro to Sci Prog with Python – Sections 3.1, 3.2, 3.4, 4.4

docs.python.org – Section 4.1 to 4.5

Iteration

- Iteration is the ability to run a block of statements repeatedly
  - In a controlled manner, choosing when to start, when to stop, or the correct number of repetitions
- New syntax:
  - `while` loops
    - repeats a block of statements as long a a condition remains `True`
    - Useful for looping an **indeterminate** number of times, until a condition is satisfied
  - `for` loops
    - Iterates through the elements of a collection or sequence (data structures and executes a block once for each elements.
    - Useful for looping a defined number of times
  - `break` to exit a loop
  - `continue` to go around again
  - `pass` to do nothing

Program control flow

- From earlier:
  - statements executed consecutively from the beginning to the end
  - An `if` statement causes branches and alternative execution

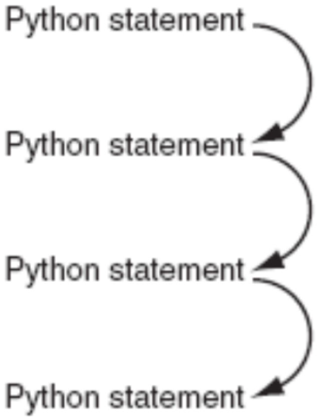


FIGURE 2.1 Sequential program flow.

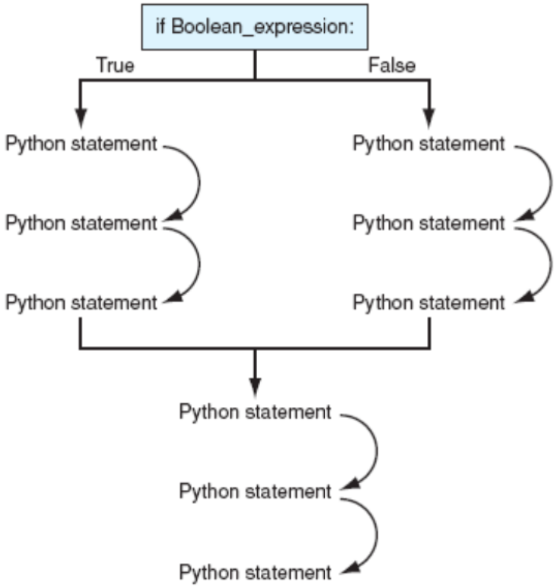


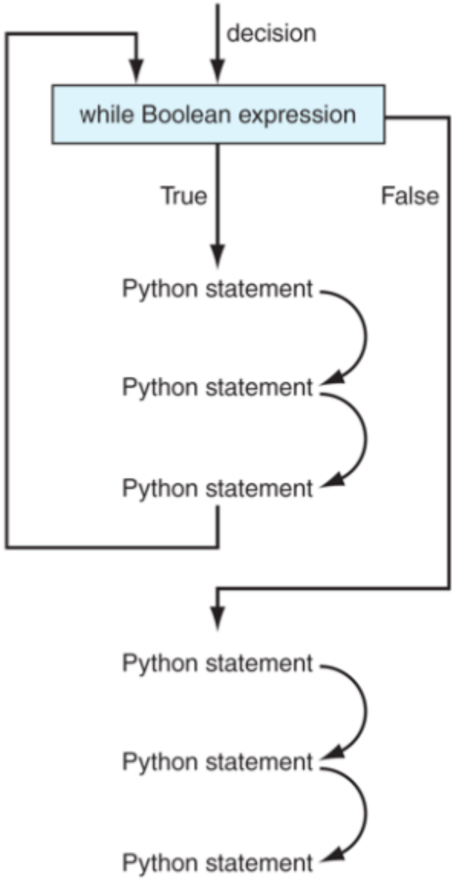
FIGURE 2.2 Decision making flow of control.

Images from Punch & Enbody

Program control flow - **iteration**

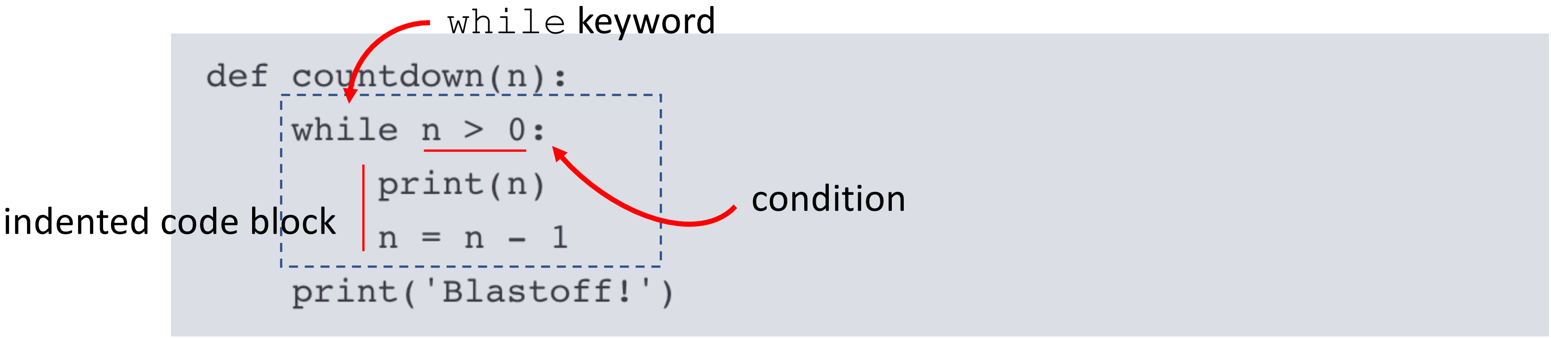
Australian National University

- Iteration repeats a block of statements
- A test is evaluated before each iteration
- The block is executed is it evaluates to `True`
- Execution will then return to the beginning of the block
- While will keep executing the block until the test statement evaluates as `False` (which may never happen…)



`while` loop syntax:

1. Starts with the `while` keyword
2. A condition, which is a statement that evaluates to `True` or `False`
3. A colon ’`:`’
4. Followed by an indented code block



```
def countdown(n):
    while n > 0:
        print(n)
        n = n - 1
    print('Blastoff!')
```

`while` keyword

condition

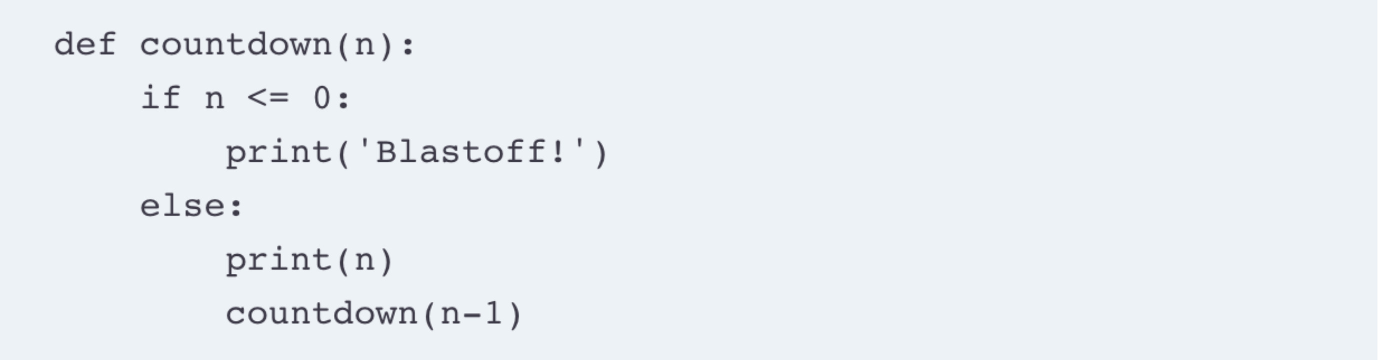
indented code block

Downey (2015) Think Python, 2nd Ed.

- This code will repeat infinitely, so long at the condition remains `True`

Example:

- Countdown example with `while`
- Using recursion:



```
def countdown(n):
    if n <= 0:
        print('Blastoff!')
    else:
        print(n)
        countdown(n-1)
```

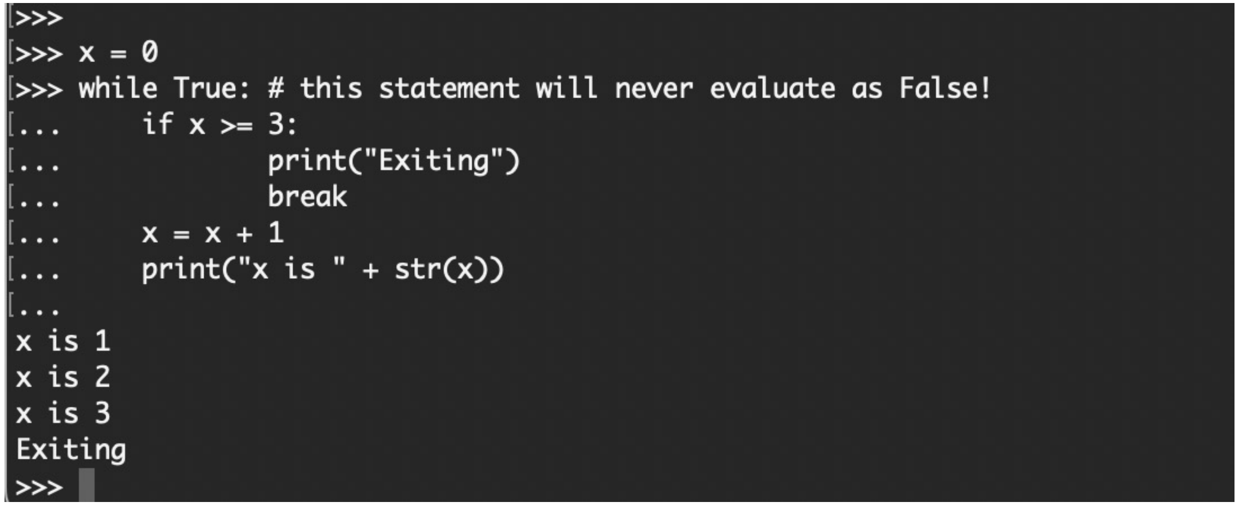
- With a `while` loop, this example is now trivially easy compared to using recursion

Another while loop example:

- Brute force compute the maximum k such that (1+2+…+k) <= 20

Exiting a `while` loop: `break`

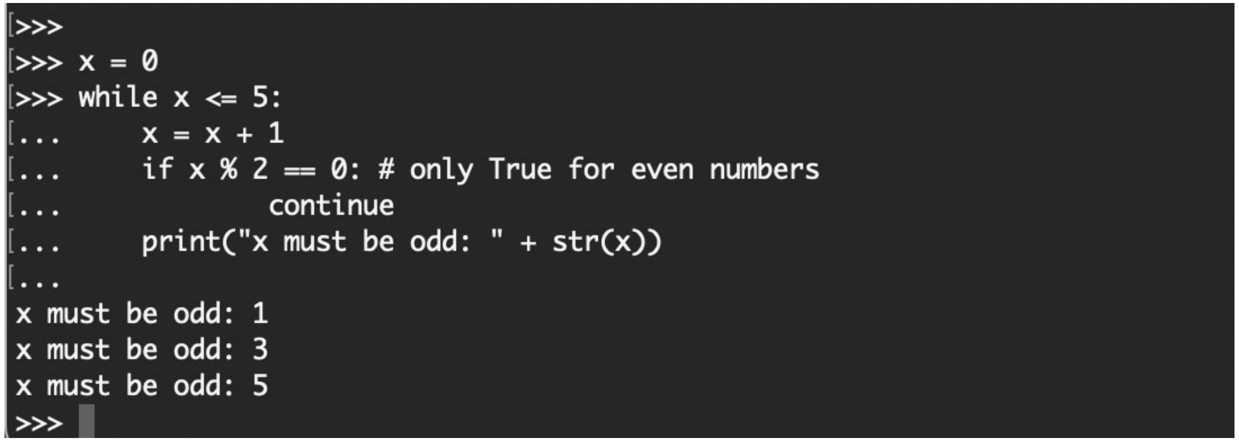
- Sometimes it is useful to exit a loop before the original `while` statement evaluates as `False`
- The `break` statement causes execution to exit and loop. Execution will re-commence from the next line of code following the end of the `while` block:



```
[>>>
[>>> x = 0
[>>> while True: # this statement will never evaluate as False!
[...     if x >= 3:
[...             print("Exiting")
[...             break
[...     x = x + 1
[...     print("x is " + str(x))
[...
x is 1
x is 2
x is 3
Exiting
>>>
```

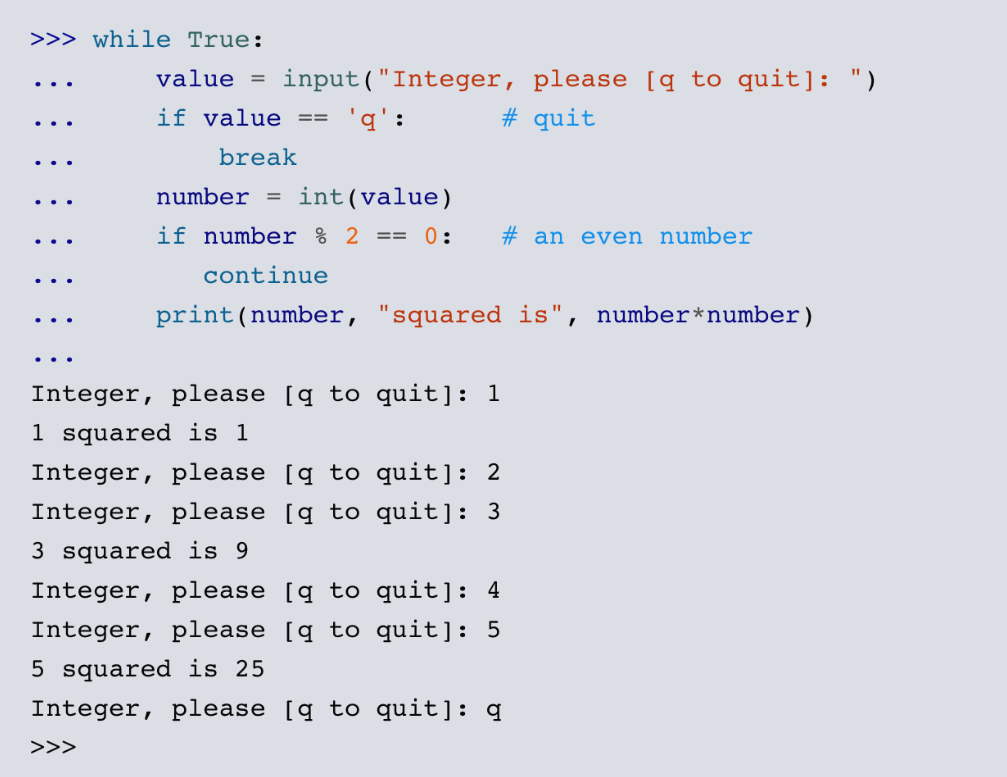
Skipping an iteration with `continue`

- And sometimes it is useful to just skip over one iteration of a loop
  - Mostly due to a condition that applies only to some iterations of the loop
- The `continue` statement causes execution to skip over the rest of the code in the while AND re-commence at the top of the code block, re-evaluating the `while` statement:



```
[>>>
[>>> x = 0
[>>> while x <= 5:
[...     x = x + 1
[...     if x % 2 == 0: # only True for even numbers
[...             continue
[...     print("x must be odd: " + str(x))
[...
x must be odd: 1
x must be odd: 3
x must be odd: 5
>>> 
```

while code example…

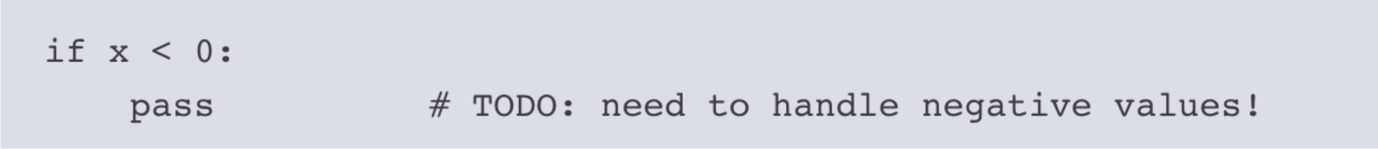


```
>>> while True:
...     value = input("Integer, please [q to quit]: ")
...     if value == 'q':      # quit
...         break
...     number = int(value)
...     if number % 2 == 0:   # an even number
...        continue
...     print(number, "squared is", number*number)
...
Integer, please [q to quit]: 1
1 squared is 1
Integer, please [q to quit]: 2
Integer, please [q to quit]: 3
3 squared is 9
Integer, please [q to quit]: 4
Integer, please [q to quit]: 5
5 squared is 25
Integer, please [q to quit]: q
>>>
```

Lubanovic (2019) *Introducing Python,* 2nd Ed.

Doing nothing with `pass`

- There is no upper limit to the number of statements that can appear in a code block
  - However, there has to be at least one statement
  - Sometimes it makes sense to have a body with no statements (or you haven’t yet implemented a function, but want to test the rest of the code)
- Use `pass` as a statement:



```
if x < 0:
    pass            # TODO: need to handle negative values!
```

- `pass` does nothing, but takes up a line

Sequences with iterators

- Strings and Lists are Sequences in Python
- hello_world = “Hello, world!”

	H	e	l	l	o	,		w	o	r	l	d	!
Element:	0	1	2	3	4	5	6	7	8	9	10	11	12

Example:

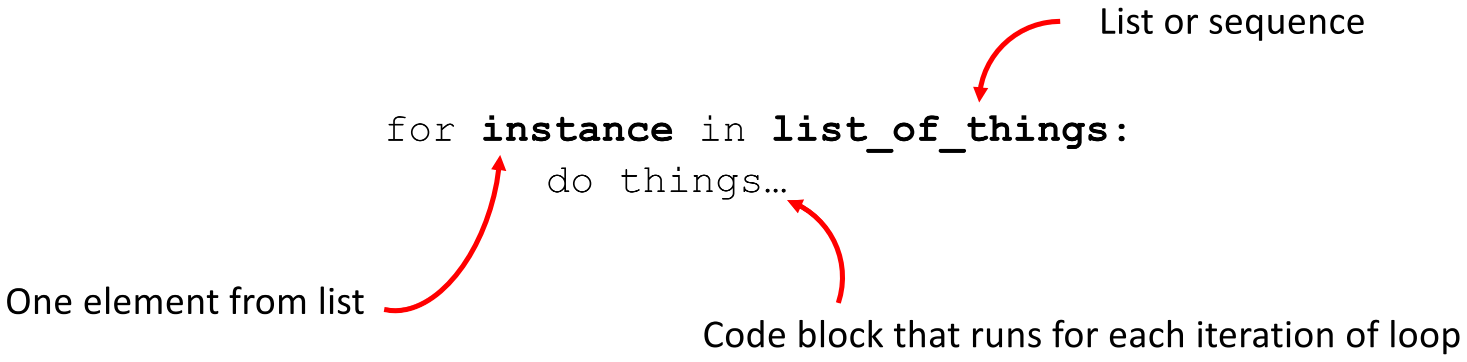
- Traversing a string with a `while` loop

Example:

- Traversing a string with a `for` loop

`for` statement syntax

- Think of these like a `while` statement, but instead of an expression evaluating truth, a `for` statement has a list to work through:



List or sequence

```
for instance in list_of_things:
      do things…
```

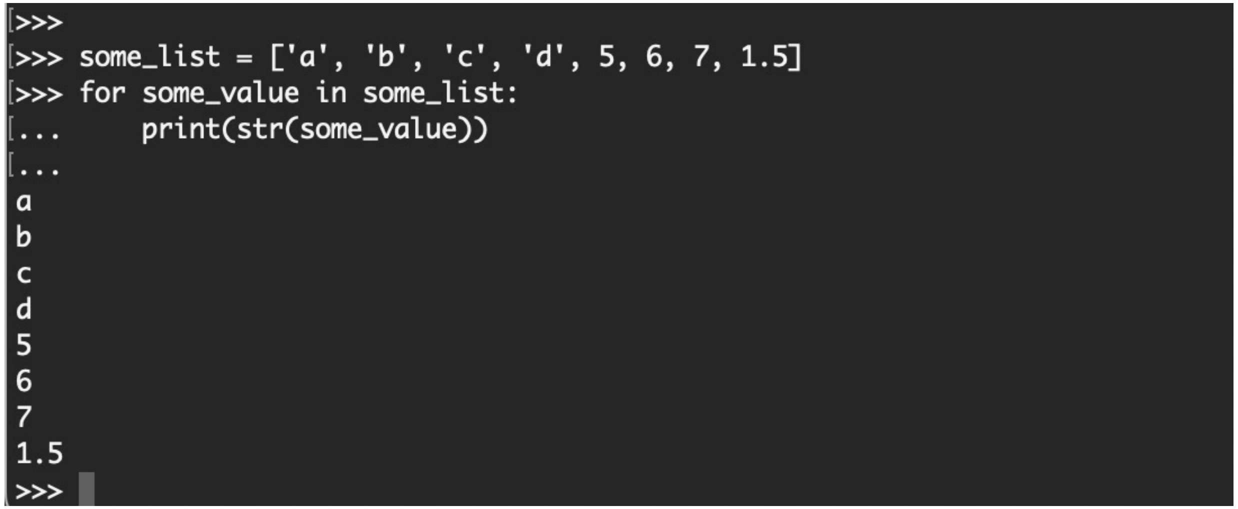
One element from list

Code block that runs for each iteration of loop

- `for` loops are perfect for iterating through lists of values

Iteration: the `for` statement

- `for` loops are bounded – meaning they have a start and an end
  - Less scary than `while`, which in unbounded and can be an infinite loop
- If we have a list, we can easily iterate through it with `for` and `in`:

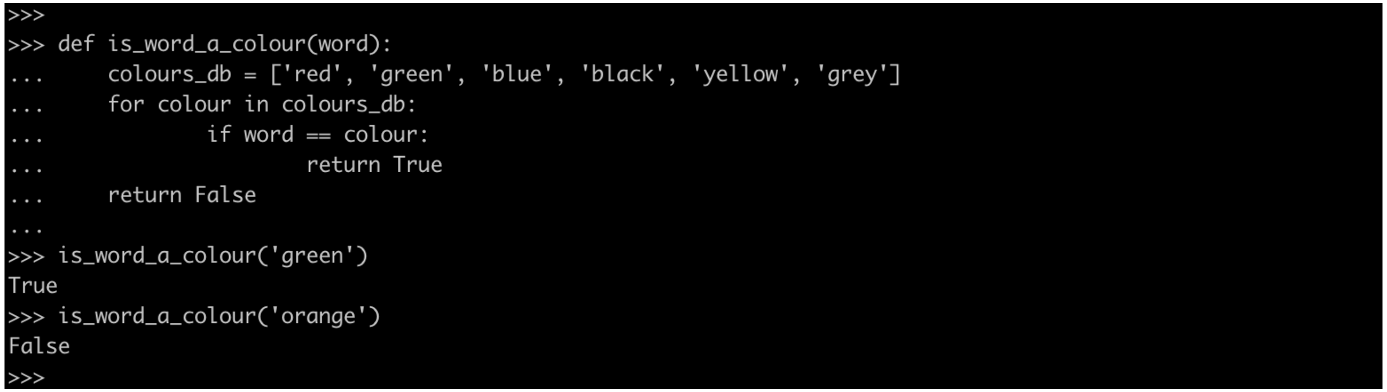


```
[>>>
[>>> some_list = ['a', 'b', 'c', 'd', 5, 6, 7, 1.5]
[>>> for some_value in some_list:
[...     print(str(some_value))
[...
a
b
c
d
5
6
7
1.5
>>>
```

- No infinite loop!

`for` loop example

- Putting together all the syntax we have learned (and a sneaky list):

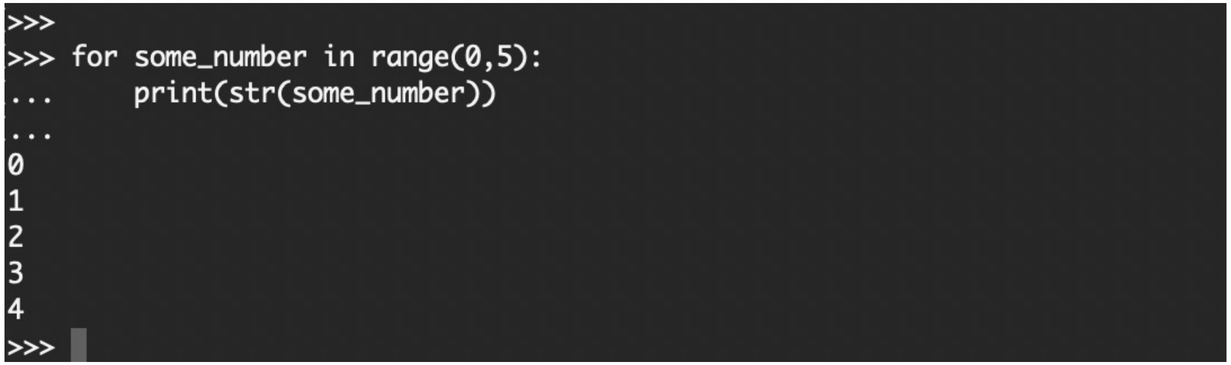


```
>>>
>>> def is_word_a_colour(word):
...     colours_db = ['red', 'green', 'blue', 'black', 'yellow', 'grey']
...     for colour in colours_db:
...             if word == colour:
...                     return True
...     return False
...
>>> is_word_a_colour('green')
True
>>> is_word_a_colour('orange')
False
>>>
```

- New syntax:
  - A `for` loop
  - Conditional execution with `if`
  - Multiple `return` statements
  - A `return` statement interrupting a `for` loop

`for` with `range()`

- It is very useful to iterate through ranges of integers.
- If you want to do something 10 times, a list containing [0,1,2,3,4,5,6,7,8,9] will let you do just this with the `for` statement.
- But having to make this list just for this is a drag – and this is just what the `range()` function does:

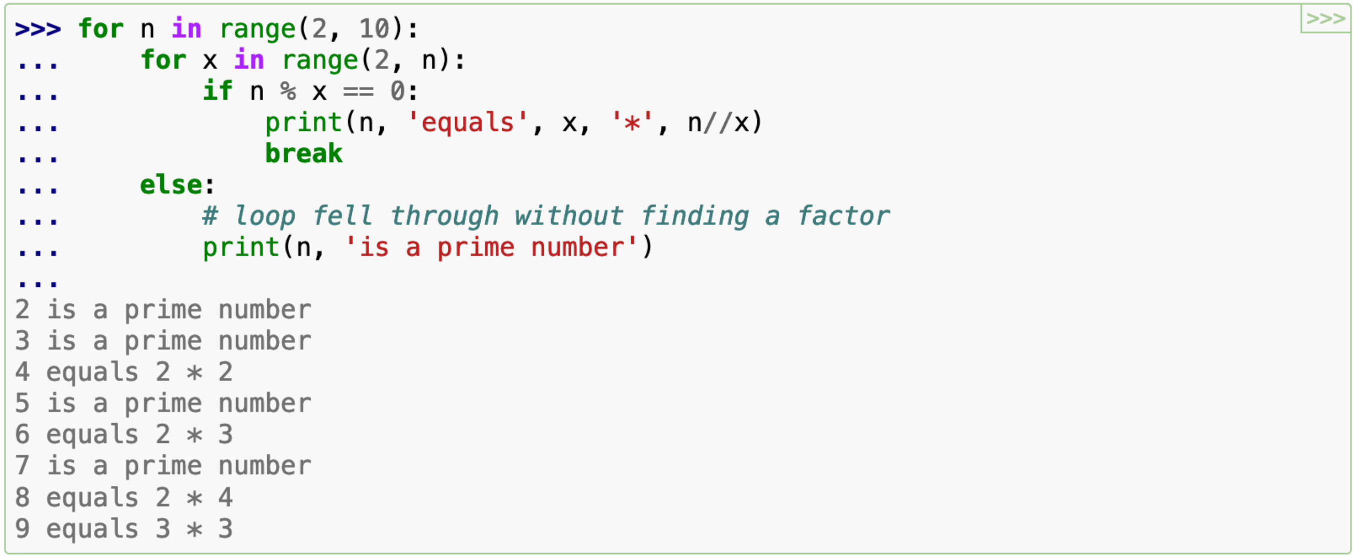


```
>>>
>>> for some_number in range(0,5):
...     print(str(some_number))
...
0
1
2
3
4
>>>
```

- It looks like `range()` returns a list – but it is more elegant than that.
- This is prosaic – and a little Python-specific - but very useful

Example

- `range()`, nested loops and `break`

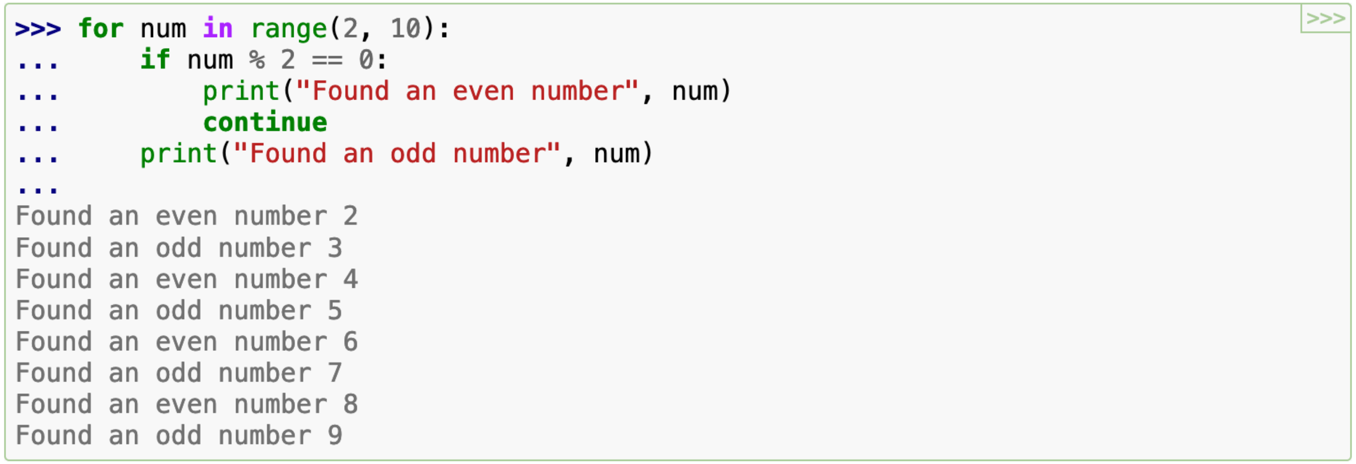


```
>>> for n in range(2, 10):
...     for x in range(2, n):
...         if n % x == 0:
...             print(n, 'equals', x, '*', n//x)
...             break
...     else:
...         # loop fell through without finding a factor
...         print(n, 'is a prime number')
...
2 is a prime number
3 is a prime number
4 equals 2 * 2
5 is a prime number
6 equals 2 * 3
7 is a prime number
8 equals 2 * 4
9 equals 3 * 3
```

Example

Australian
National
University

- `range()` and `continue`



```
>>> for num in range(2, 10):
...     if num % 2 == 0:
...         print("Found an even number", num)
...         continue
...     print("Found an odd number", num)
...
Found an even number 2
Found an odd number 3
Found an even number 4
Found an odd number 5
Found an even number 6
Found an odd number 7
Found an even number 8
Found an odd number 9
```

Exercises

- Complete Exercises 10-1 and 10-2 of *Think Python*.
- And (if you liked Ch 7) Exercises 7-1, 7-2 and 7-3 of *Think Python*.

Reading

- Chapter 7 (very brief chapter) of *Think Python, ‘the while statement*’ and ’*break*’
- Chapter 10 (first three sections, including *Traversing a List*) of *Think Python*