C10552: Intro to Computation

Lecture 2
July 17, 2016

Lecture 1 Recap

- Loops are useful for repeating code

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

- If statements are useful for conditional code

```
if current_temp > 80:
    print "It's really hot out there!"
```

Lecture 2 Overview

- HW solutions
- One more type of loop: **`while`**
    - “Run while condition is true”
- Lists
- Functions
- Live coding

Homework: finding prime divisors

The following paragraph is an excerpt from a book published in 1874:

Can the reader say what two numbers multiplied together will produce the number 8,616,460,799? I think it unlikely that anyone but myself will ever know; for they are two large prime numbers, and can only be rediscovered by trying in succession a long series of prime divisors until the right one be fallen upon.

Find these two numbers.

Homework: finding prime divisors

Plan for the code:

- Use the writer's suggestion: try every possible number!

Homework: finding prime divisors

Plan for the code:

- Use the writer's suggestion: try every possible number!
- If any number less than 8,616,460,799 divides it evenly, print it.

Homework: finding prime divisors

Plan for the code:

- Use the writer’s suggestion: try every possible number!
- If any number less than 8,616,460,799 divides it evenly, print it.

- We’ll need…

Homework: finding prime divisors

Plan for the code:

- Use the writer’s suggestion: try every possible number!
- If any number less than 8,616,460,799 divides it evenly, print it.

- We’ll need… **a for loop, an if statement, and the modulo operator**

Live coding!

Plan for the code:

- Use the writer's suggestion: try every possible number!
- If any number less than 8,616,460,799 divides it evenly, print it.

- We'll need… **a for loop, an if statement, and the modulo operator**

Live coding!

```
>>> for i in xrange(1, 8616460799):
...     if 8616460799 % i is 0:
...         print i
...
1
89681
96079
```

The while loop

The while loop

- “Run while condition is true”

```
red_sox = 2; yankees = 1
while red_sox > yankees:
    print “go sox” # runs forever!
```

Condition is checked *before* each iteration.

The while loop

- “Run while condition is true”

```
red_sox = 2; yankees = 1
while red_sox > yankees:
    print “go sox” # runs forever!
```

Anything after # is a comment.
It is for *your own eyes.*
The computer will ignore it.

The while loop

- “Run while condition is true”

Anything after # is a comment.
It is for *your own eyes.*
The computer will ignore it.

```
red_sox = 2; yankees = 1
while red_sox > yankees:
    print “go sox” # runs forever!
```

- While loops do not have a predetermined number of iterations. (unlike for loops)

The while loop: example

- What is the output of the following code?

```
n = 10
while n < 100:
    n = n * 2
    print n
```

The while loop: example

- What is the output of the following code?

```
n = 10
while n < 100:
    n = n * 2
    print n
```

```
20
40
80
160
```

The while loop: example

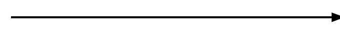
- What is the output of the following code?

```
n = 10
while n < 100:
    n = n * 2
    print n
```

20

40

80

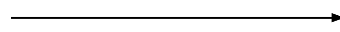
160  what?

The while loop: example

- What is the output of the following code?

```
n = 10
while n < 100:
    n = n * 2
    print n
```

```
20
40
80
160
```



what?

Remember that the condition is checked **before** each iteration!

Lists

- One of the most important tools in programming
- Simple syntax in Python:

```
my_list = [1,2,3,4,5]
```

- range(...) is actually a list!

```
>>> print range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

- You can *iterate* over lists

Lists

- You can *iterate* over lists

```
>>> my_friends = ["jim", "al", "mary"]
>>> for friend in my_friends:
...     print "hello", friend
...
hello jim
hello al
hello mary
```

Lists

- You can *iterate* over lists

```
>>> my_friends = ["jim", "al", "mary"]
>>> for friend in my_friends:
...     print "hello", friend
...
hello jim
hello al
hello mary
```

If you are not iterating over `range(...)`, use a descriptive name instead of `i`

Lists

- You can *access* list elements

```
>>> my_friends = ["jim", "al", "mary"]
>>> print my_friends[0]
jim
>>> print my_friends[1]
al
>>> print my_friends[2]
mary
```

Modifying lists

- You can modify elements in-place

```
>>> my_friends = ["jim", "al", "mary"]
>>> my_friends[0] = “joe”
>>> my_friends
['joe', 'al', 'mary']
```

Modifying lists

- You can append to lists

```
>>> my_friends = ["jim", "al", "mary"]
>>> my_friends.append("alice")
>>> my_friends
['jim', 'al', 'mary', 'alice']
```

- You can add two lists to each other

```
>>> your_friends = ["jack", "mary",
"jessica"]
>>> my_friends + your_friends
['jim', 'al', 'mary', 'alice', 'jack',
'mary', 'jessica']
```

Modifying lists

- You can append to lists

```
>>> my_friends = ["jim", "al", "mary"]
>>> my_friends.append("alice")
>>> my_friends
['jim', 'al', 'mary', 'alice']
```

- You can add two lists to each other

```
>>> your_friends = ["jack", "mary",
"jessica"]
>>> my_friends + your_friends
['jim', 'al', 'mary', 'alice', 'jack',
'mary', 'jessica']
```

Repetitions are allowed!

Lists are flexible!

- They can include any type of data...

```
>>> my_favorite_numbers = ["one", 2, 42.0]
```

- ... even other lists...

```
>>> your_fav_numbers = [5, 100, 65536]
>>> our_fav_numbers = [my_fav_numbers, your_fav_numbers]
>>> our_fav_numbers
[['one', 2, 42.0], [5, 100, 65536]]
```

- ... even themselves!

```
>>> my_favorite_numbers.append(my_favorite_numbers)
>>> my_favorite_numbers
['one', 2, 42.0, [...]]
>>> my_favorite_numbers[3] is my_favorite_numbers
True
```

Lists are flexible!

- They can include any type of data...

```
>>> my_favorite_numbers = ["one", 2, 42.0]
```

- ... even other lists...

```
>>> your_fav_numbers = [5, 100, 65536]
>>> our_fav_numbers = [my_fav_numbers, your_fav_numbers]
>>> our_fav_numbers
[['one', 2, 42.0], [5, 100, 65536]]
```

- ... even themselves!

```
>>> my_favorite_numbers.append(my_favorite_numbers)
>>> my_favorite_numbers
['one', 2, 42.0, [...]]
>>> my_favorite_numbers[3] is my_favorite_numbers
True
```

Lists are flexible!

- They can include any type of data...

```
>>> my_favorite_numbers = ["one", 2, 42.0]
```

- ... even other lists...

```
>>> your_fav_numbers = [5, 100, 65536]
>>> our_fav_numbers = [my_fav_numbers, your_fav_numbers]
>>> our_fav_numbers
[['one', 2, 42.0], [5, 100, 65536]]
```

- ... even themselves!

```
>>> my_favorite_numbers.append(my_favorite_numbers)
>>> my_favorite_numbers
['one', 2, 42.0, [...]]
>>> my_favorite_numbers[3] is my_favorite_numbers
True
```

Strings: pretty much like lists

- Time to formally define strings!
- You have seen a string before: `print “hello world”`
- Most of the operations are the same as lists…

```
>>> my_string = "hello world"
>>> my_string[0]
'h'
>>> your_string = “123”
>>> my_string + your_string
'hello world123'
```

- … but a string only contains characters, and it cannot be modified in-place

```
>>> my_string[0] = 'a'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

Strings: pretty much like lists

- Time to formally define strings!
- You have seen a string before: `print “hello world”`
- Most of the operations are the same as lists…

```
>>> my_string = "hello world"
>>> my_string[0]
'h'
>>> your_string = “123”
>>> my_string + your_string
'hello world123'
```

- … but a string only contains characters, and it cannot be modified in-place

```
>>> my_string[0] = 'a'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

'str' object — i.e. string variable

Strings: pretty much like lists

- A string only contains characters, and it cannot be modified in-place

```
>>> my_string[0] = 'a'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

- You need to build a new string if you want to modify it

```
>>> english_word = "hello"
>>> german_word = ""
>>> for letter in english_word:
...     if letter is 'e':
...         german_word = german_word + "a"
...     else:
...         german_word = german_word + letter
...
>>> german_word
'hallo'
```

Functions

- We sometimes have to do the same set of things over, and over, and over again...
- Functions are a great way to “pack” your code

```
def my_function(input_value):
# Multiplies input_value by 2.
    return input_value * 2
```

Functions

- We sometimes have to do the same set of things over, and over, and over again...
- Functions are a great way to “pack” your code

```
def my_function(input_value):
# Multiplies input_value by 2.
    return input_value * 2
```

Treat this as a black box.

Functions

- Abstraction: very important concept in coding
- don’t reinvent the wheel!

```
def shift_letter(letter):
# Given a letter, returns the next
# letter in the alphabet.
    return chr(ord(letter)+1)
```

Functions

- Abstraction: very important concept in coding
- don't reinvent the wheel!

```
def shift_letter(letter):
# Given a letter, returns the next
# letter in the alphabet.
    return chr(ord(letter)+1)
```

You may have no idea what the red outlined text means. That is fine. You only need to know what this function does.

Functions

- Abstraction: very important concept in coding
- don't reinvent the wheel!

```
def shift_letter(letter):
# Given a letter, returns the next
# letter in the alphabet.
    return chr(ord(letter)+1)
```

You may have no idea what the red outlined text means. That is fine. You only need to know what this function does.

Functions

- Let's test our black box...

```
>>> shift_letter('a')
'b'
>>> shift_letter('v')
'w'
>>> shift_letter('z')
'{'
>>> shift_letter('d')
'e'
>>> shift_letter('m')
'n'
```

Functions

- Let's test our black box...

```
>>> shift_letter('a')
'b'
>>> shift_letter('v')
'w'
>>> shift_letter('z')
'{'
>>> shift_letter('d')
'e'
>>> shift_letter('m')
'n'
```

Functions

- Let’s test our black box…

```
>>> shift_letter('a')
'b'
>>> shift_letter('v')
'w'
>>> shift_letter('z')
'{'
>>> shift_letter('d')
'e'
>>> shift_letter('m')
'n'
```

Issue: Function does not return a letter for ‘z’

Functions

- Let’s test our black box…

```
>>> shift_letter('a')
'b'
>>> shift_letter('v')
'w'
>>> shift_letter('z')
'{'
>>> shift_letter('d')
'e'
>>> shift_letter('m')
'n'
```

Issue: Function does not return a letter for ‘z’

Solution: Modify function so that ‘z’ maps to ‘a’

Functions

```
def shift_letter(letter):
# Given a letter, returns the next
# letter in the alphabet.
    return chr(ord(letter)+1)
```

Functions

```
def shift_letter(letter):
# Given a letter, returns the next
# letter in the alphabet.
    return chr(ord(letter)+1)
```

```
def shift_letter(letter):
# Given a letter, returns the next
# letter in the alphabet.
    return chr((ord(letter)-97+1)%26+97)
```

Exercise: understand how this function works.

Hint: ord('a') is equal to 97.

Solution at the end.

Functions

```
>>> shift_letter('a')
'b'
>>> shift_letter('b')
'c'
>>> shift_letter('z')
'a'
>>> shift_letter('w')
'x'
>>> shift_letter('x')
'y'
```

Functions

```
>>> shift_letter('a')
'b'
>>> shift_letter('b')
'c'
>>> shift_letter('z')
'a'
>>> shift_letter('w')
'x'
>>> shift_letter('x')
'y'
```

All set!

Live coding!

Using shift_letter, write a simple encryption algorithm that shifts each letter by a given number (*not necessarily 1*). Assume the input text is lowercase.

```
def encrypt(input_text, shift):
    # Shifts each letter in input_text
    # by the given shift value.
    # Returns the new string.
```

We’re done!

See you next week!

Solution to Exercise

```
def shift_letter(letter):
    # ord(...) returns 97 for 'a'.
    # Subtract that from letter
    # to get the alphabetical rank of letter.
    # Note that this will be using zero-indexing
    # (i.e. 'a' will be 0)
    alpha_rank = ord(letter) - 97
    # shift by 1
    new_alpha_rank = alpha_rank + 1
    # use the modulo operator so 26 maps to 0
    new_alpha_rank = new_alpha_rank % 26
    # ord() and chr() are inverses of each other.
    # Add the subtracted 97 back and return using chr.
    return chr(new_alpha_rank + 97)
```