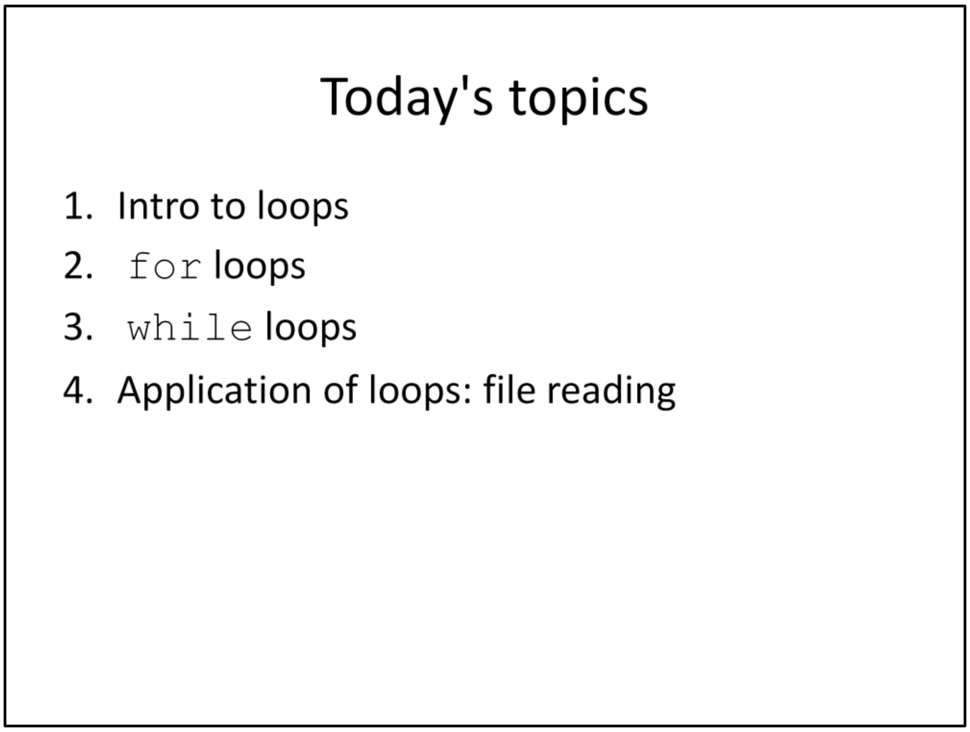
Control flow and logic, part II: `for` and `while` loops

Programming Bootcamp, 2015

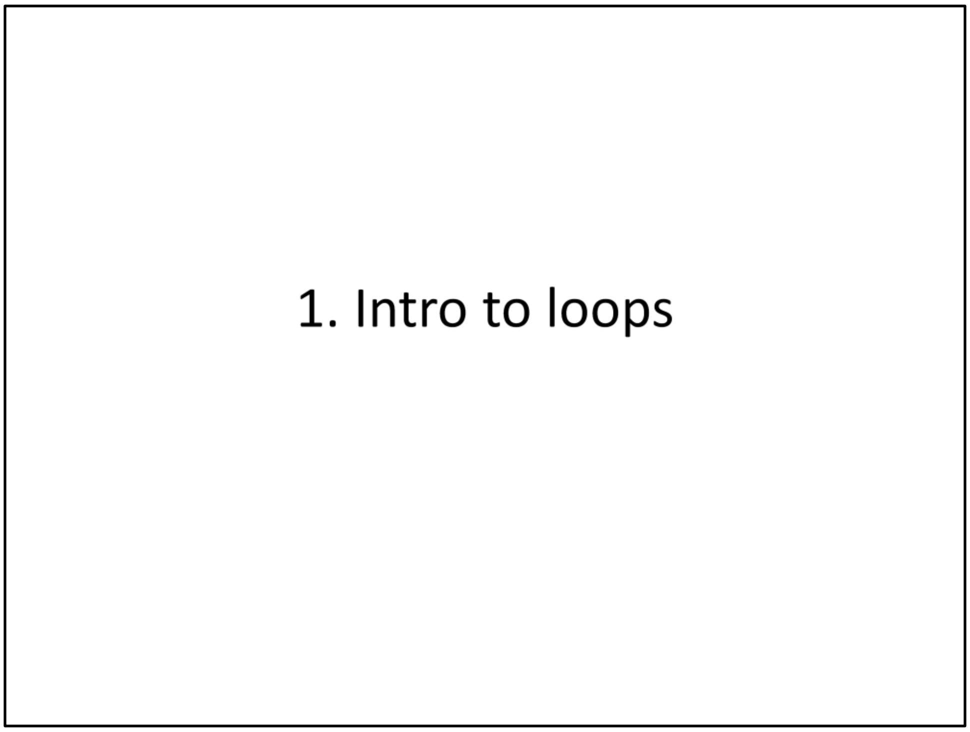
Day 3 - 6/9/15

Remember to sign in!

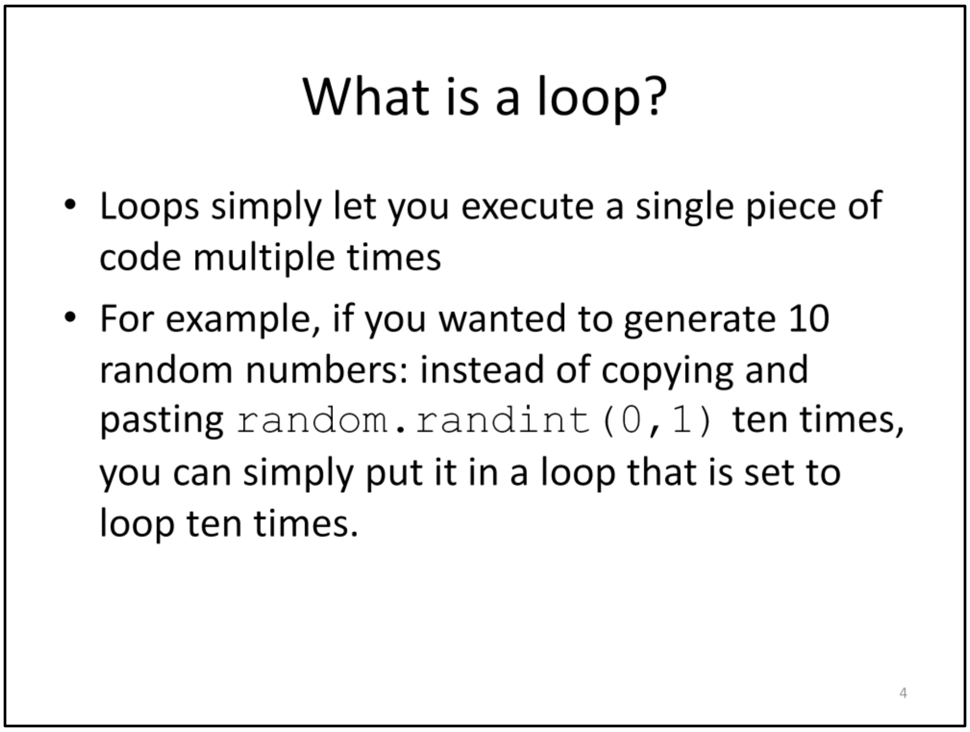


Today's topics

1. Intro to loops
2. `for` loops
3. `while` loops
4. Application of loops: file reading



1. Intro to loops

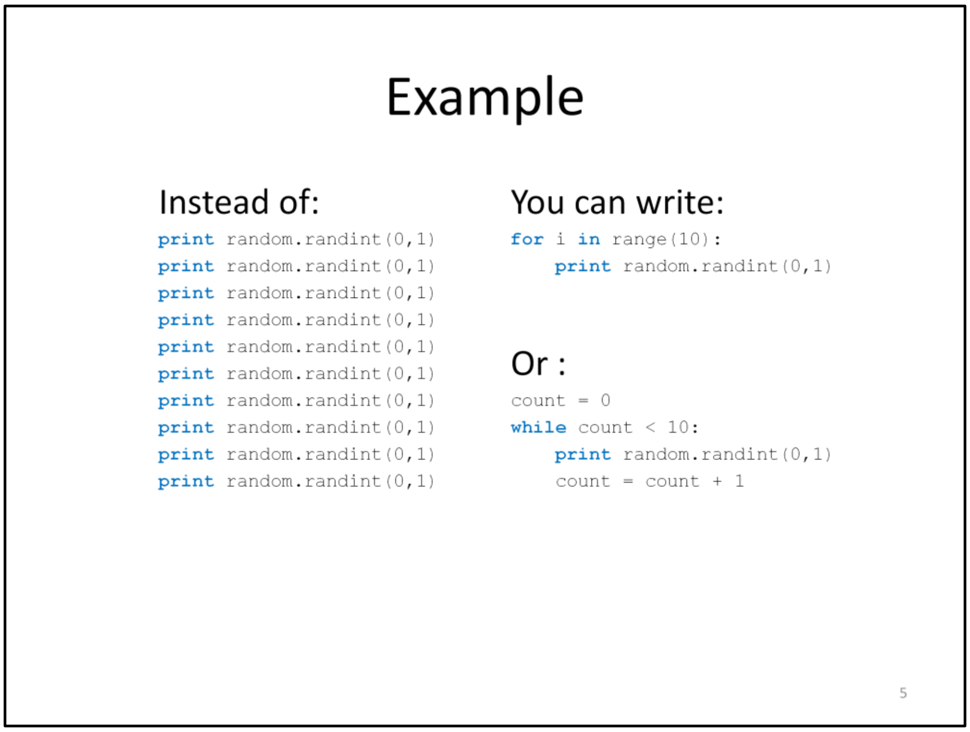


What is a loop?

- Loops simply let you execute a single piece of code multiple times
- For example, if you wanted to generate 10 random numbers: instead of copying and pasting `random.randint(0,1)` ten times, you can simply put it in a loop that is set to loop ten times.

4

Sorry for that annoying problem set problem last time, but I wanted you to really feel the need for loops! ☺



Example

Instead of:

```
print random.randint(0,1)
print random.randint(0,1)
print random.randint(0,1)
print random.randint(0,1)
print random.randint(0,1)
print random.randint(0,1)
print random.randint(0,1)
print random.randint(0,1)
print random.randint(0,1)
print random.randint(0,1)
```

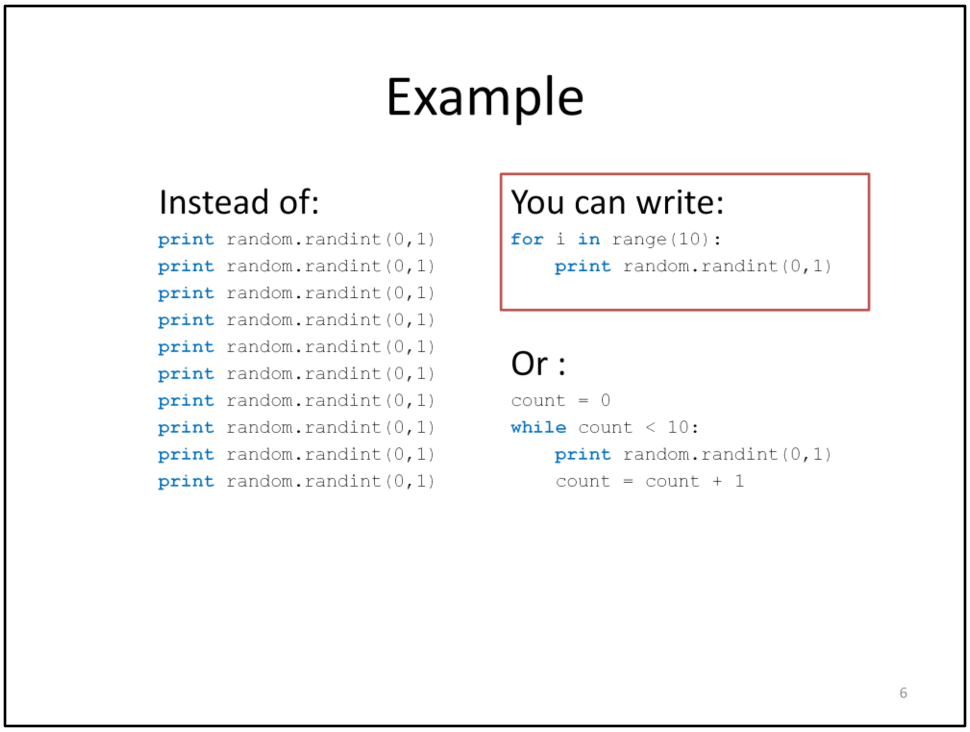
You can write:

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

Or :

```
count = 0
while count < 10:
    print random.randint(0,1)
    count = count + 1
```

5



Example

Instead of:

```
print random.randint(0,1)
print random.randint(0,1)
print random.randint(0,1)
print random.randint(0,1)
print random.randint(0,1)
print random.randint(0,1)
print random.randint(0,1)
print random.randint(0,1)
print random.randint(0,1)
print random.randint(0,1)
```

You can write:

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

Or :

```
count = 0
while count < 10:
    print random.randint(0,1)
    count = count + 1
```

6

First we'll talk about for loops.

2. `for` loops

The `for` loop

Purpose: execute a block of code a specific number of times.

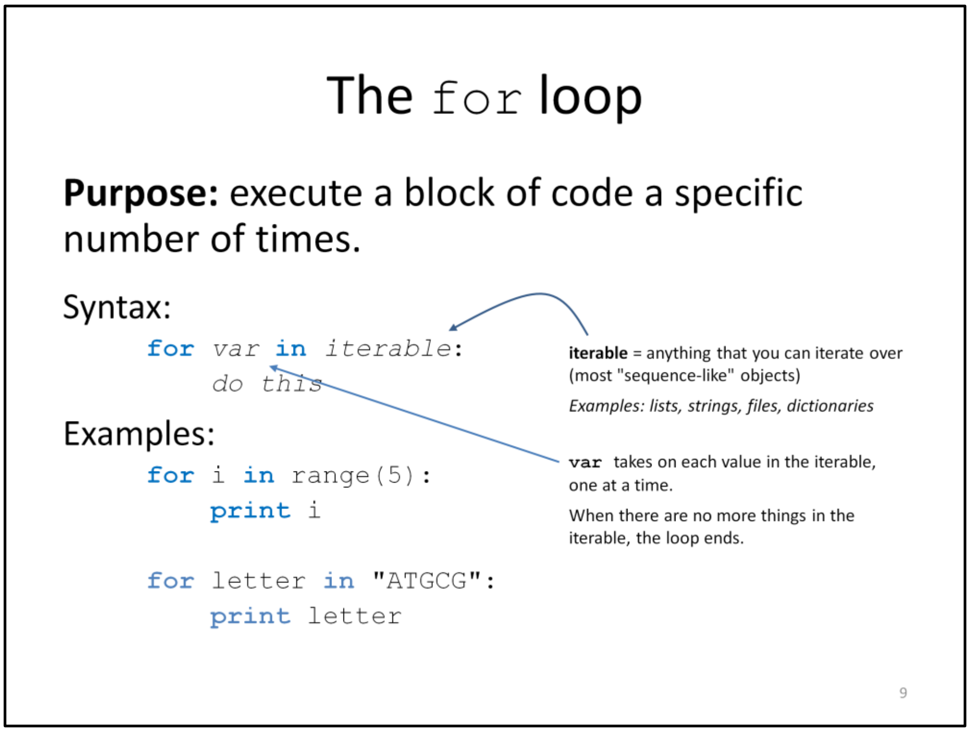
Syntax:

```
for var in iterable:
    do this
```

Examples:

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

for letter in "ATGCG":
    print letter
```

8



The `for` loop

Purpose: execute a block of code a specific number of times.

Syntax:

```
for var in iterable:
    do this
```

iterable = anything that you can iterate over (most "sequence-like" objects)

Examples: lists, strings, files, dictionaries

`var` takes on each value in the iterable, one at a time.

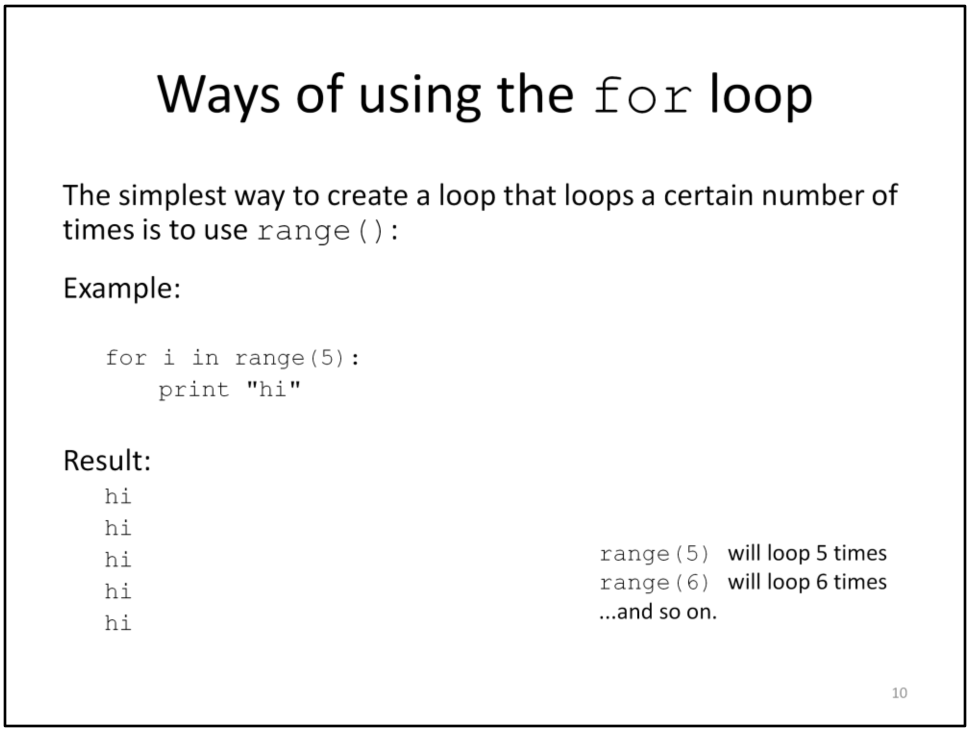
When there are no more things in the iterable, the loop ends.

Examples:

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

for letter in "ATGCG":
    print letter
```

9

Note the indentation, much like with if/else statements, is very important and defines what belongs to the for loop.



Ways of using the `for` loop

The simplest way to create a loop that loops a certain number of times is to use `range()`:

Example:

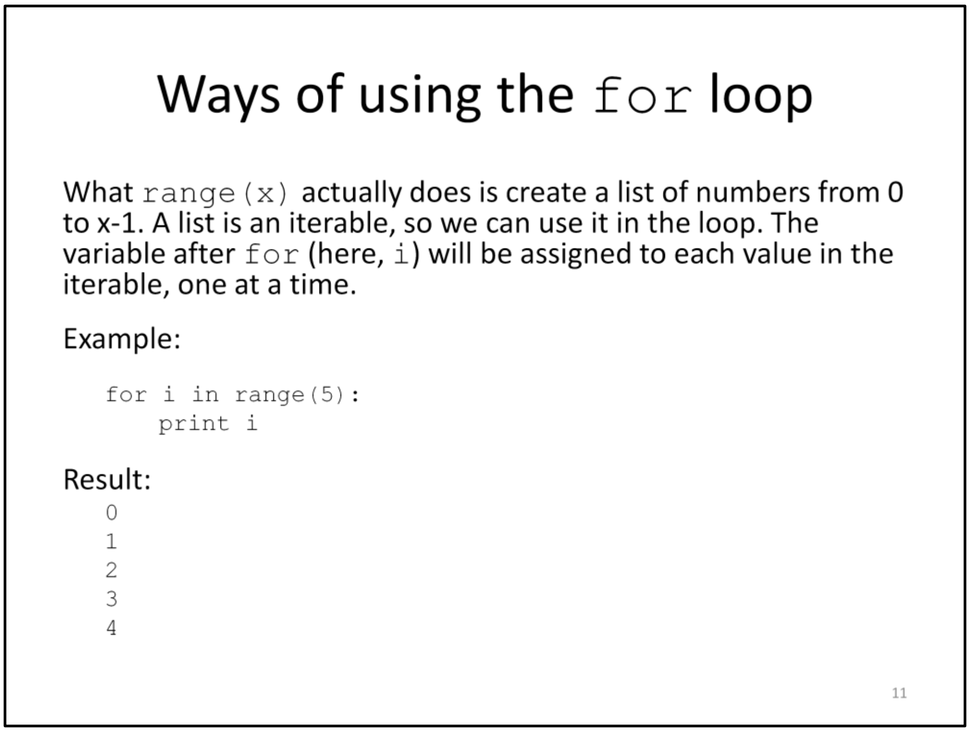
```
for i in range(5):
    print "hi"
```

Result:

```
hi
hi
hi
hi
hi
```

`range(5)` will loop 5 times
`range(6)` will loop 6 times
...and so on.

10



Ways of using the `for` loop

What `range(x)` actually does is create a list of numbers from 0 to x-1. A list is an iterable, so we can use it in the loop. The variable after `for` (here, `i`) will be assigned to each value in the iterable, one at a time.

Example:

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

Result:

```
0
1
2
3
4
```

11

More about how range() works in a minute, and more on lists next time.

Ways of using the `for` loop

A string is also an iterable, and so we can use a `for` loop to iterate over each individual character in the string, one at a time:

Example:

```
for letter in "Hello!":
    print letter
```

Result:

```
H
e
l
l
o
!
```

Important to note:

You can name the variable after for anything you want, and you do NOT need to define it before using it in the for loop.

Practice with `for`

What will the following code print?

```
for i in range(4):
    print i
```

Practice with `for`

What will the following code print?

```
for i in range(4):
    print i
```

Result:

```
0
1
2
3
```

Practice with `for`

What will the following code print?

```
for i in range(4):
    print i * 2
```

Practice with `for`

What will the following code print?

```
for i in range(4):
    print i * 2
```

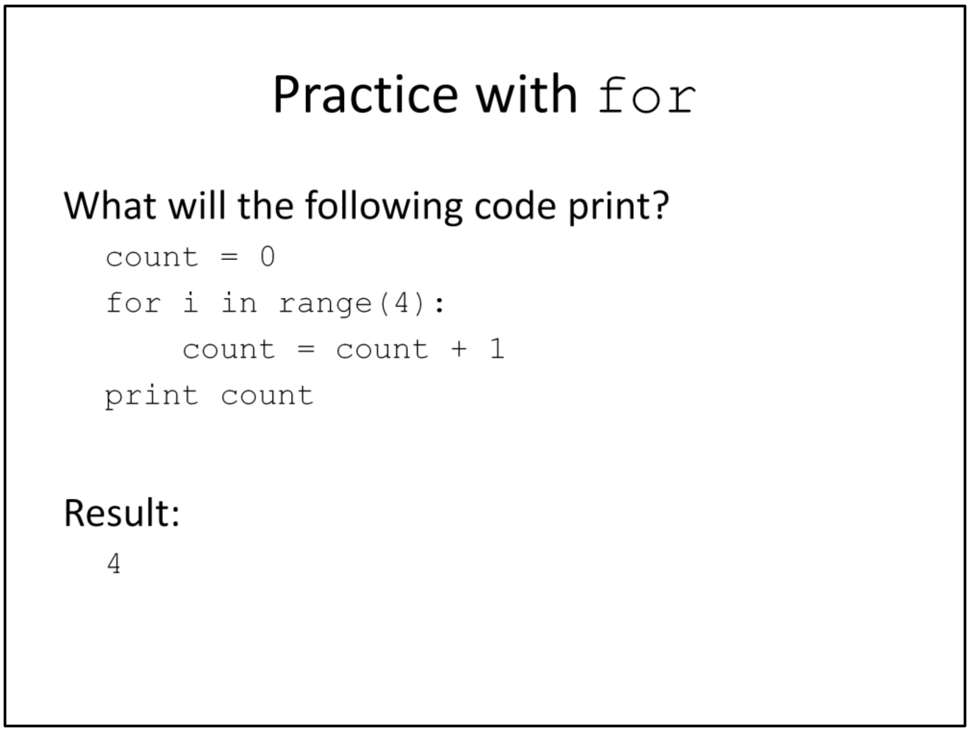
Result:

```
0
2
4
6
```

Practice with `for`

What will the following code print?

```
count = 0
for i in range(4):
    count = count + 1
print count
```



Practice with `for`

What will the following code print?

```
count = 0
for i in range(4):
    count = count + 1
print count
```

Result:

```
4
```

This is a basic counter. Kind of pointless in this particular example, but overall a very very useful thing. Remember it!

Practice with `for`

What will the following code print?

```
count = 0
for i in range(4):
    count = count + i
print count
```

Practice with `for`

What will the following code print?

```
count = 0
for i in range(4):
    count = count + i
print count
```

Result:

```
6
```

Important to note:

This is similar to a counter, but instead of adding 1 each time, we're adding up various numbers.

This is sometimes called an *accumulator*, and it's useful in many situations, so remember it!

Practice with `for`

What will the following code print?

```
for nt in "ATGAT":
    print nt
```

Practice with `for`

What will the following code print?

```
for nt in "ATGAT":
    print nt
```

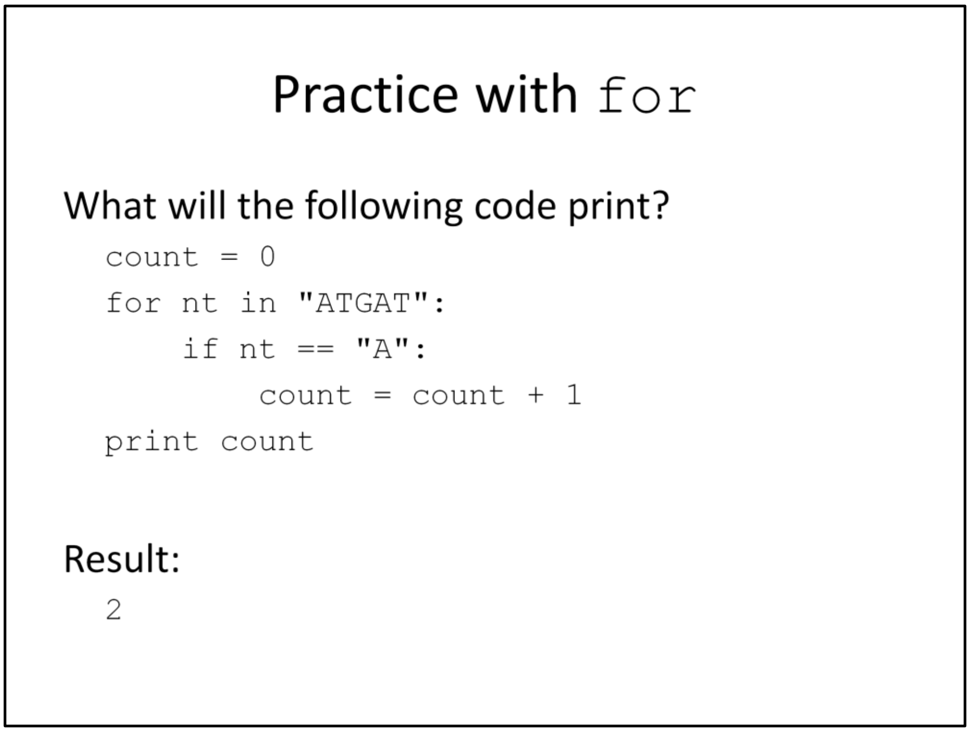
Result:

```
A
T
G
A
T
```

Practice with `for`

What will the following code print?

```
count = 0
for nt in "ATGAT":
    if nt == "A":
        count = count + 1
print count
```


Practice with `for`

What will the following code print?

```
count = 0
for nt in "ATGAT":
    if nt == "A":
        count = count + 1
print count
```

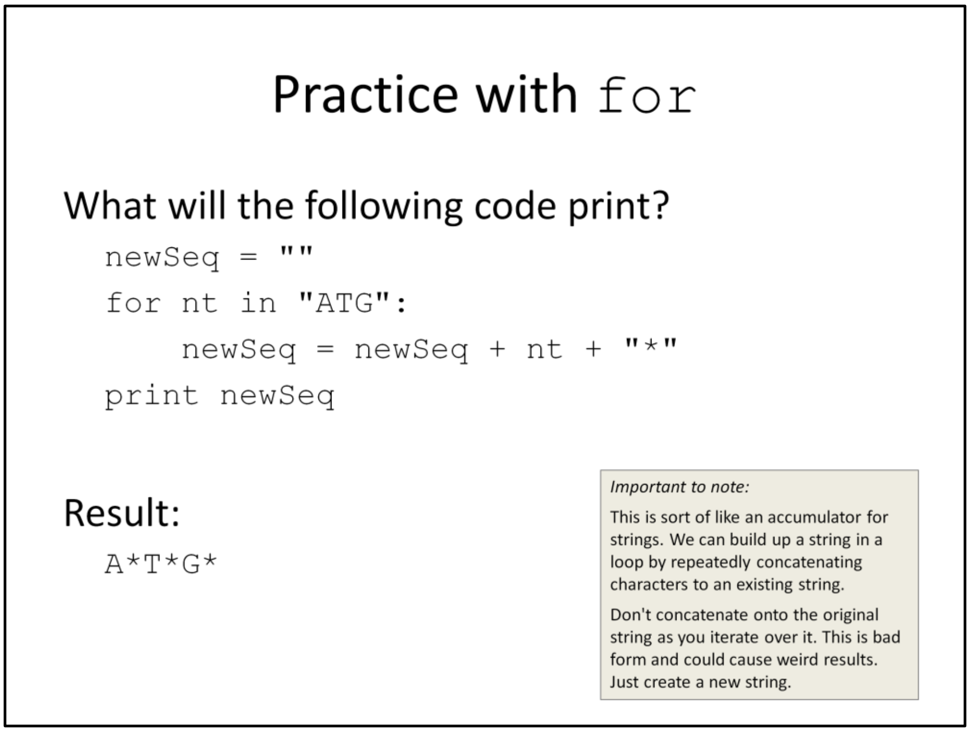
Result:

```
2
```

Practice with `for`

What will the following code print?

```
newSeq = ""
for nt in "ATG":
    newSeq = newSeq + nt + "*"
print newSeq
```



Practice with `for`

What will the following code print?

```
newSeq = ""
for nt in "ATG":
    newSeq = newSeq + nt + "*"
print newSeq
```

Result:

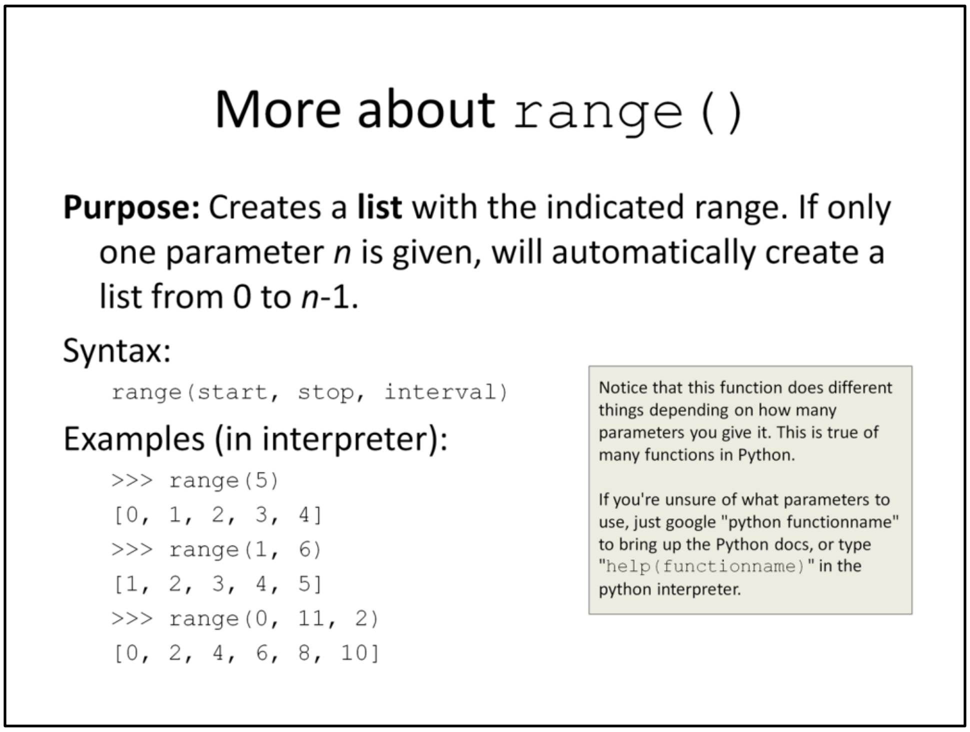
```
A*T*G*
```

Important to note:

This is sort of like an accumulator for strings. We can build up a string in a loop by repeatedly concatenating characters to an existing string.

Don't concatenate onto the original string as you iterate over it. This is bad form and could cause weird results. Just create a new string.

Kind of a dumb example, but a very important concept.



More about `range()`

Purpose: Creates a **list** with the indicated range. If only one parameter *n* is given, will automatically create a list from 0 to *n*-1.

Syntax:

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

Examples (in interpreter):

```
>>> range(5)
[0, 1, 2, 3, 4]
>>> range(1, 6)
[1, 2, 3, 4, 5]
>>> range(0, 11, 2)
[0, 2, 4, 6, 8, 10]
```

Notice that this function does different things depending on how many parameters you give it. This is true of many functions in Python.

If you're unsure of what parameters to use, just google "python functionname" to bring up the Python docs, or type "`help(functionname)`" in the python interpreter.

It may seem a little odd that it goes from 0 to n-1, but since most things in programming start counting at 0, it ends up being more convenient this way.

We'll learn more about lists next time!

Practice with `range()`

What will the following code print?

```
print range(4)
```

Practice with `range()`

What will the following code print?

```
print range(4)
```

Result:

```
[0, 1, 2, 3]
```

Practice with `range()`

What will the following code print?

```
print range(4, 8)
```

Practice with `range()`

What will the following code print?

```
print range(4, 8)
```

Result:

```
[4, 5, 6, 7]
```

Practice with `range()`

What will the following code print?

```
print range(0, 50, 10)
```

Practice with `range()`

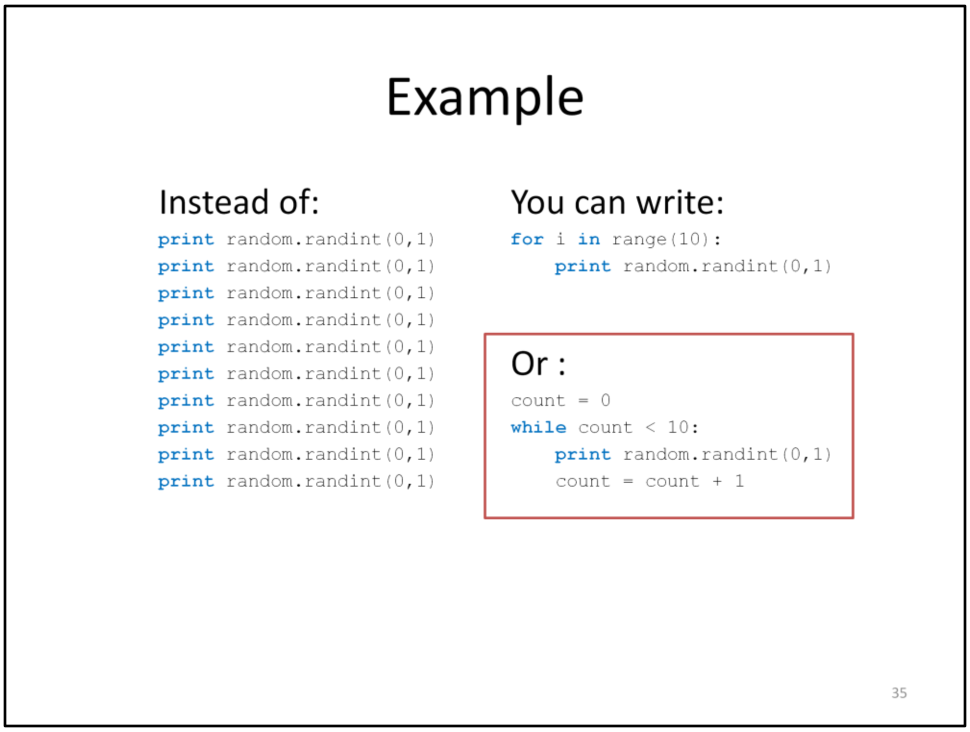
What will the following code print?

```
print range(0, 50, 10)
```

Result:

```
[0, 10, 20, 30, 40]
```

3. `while` loops



Example

Instead of:

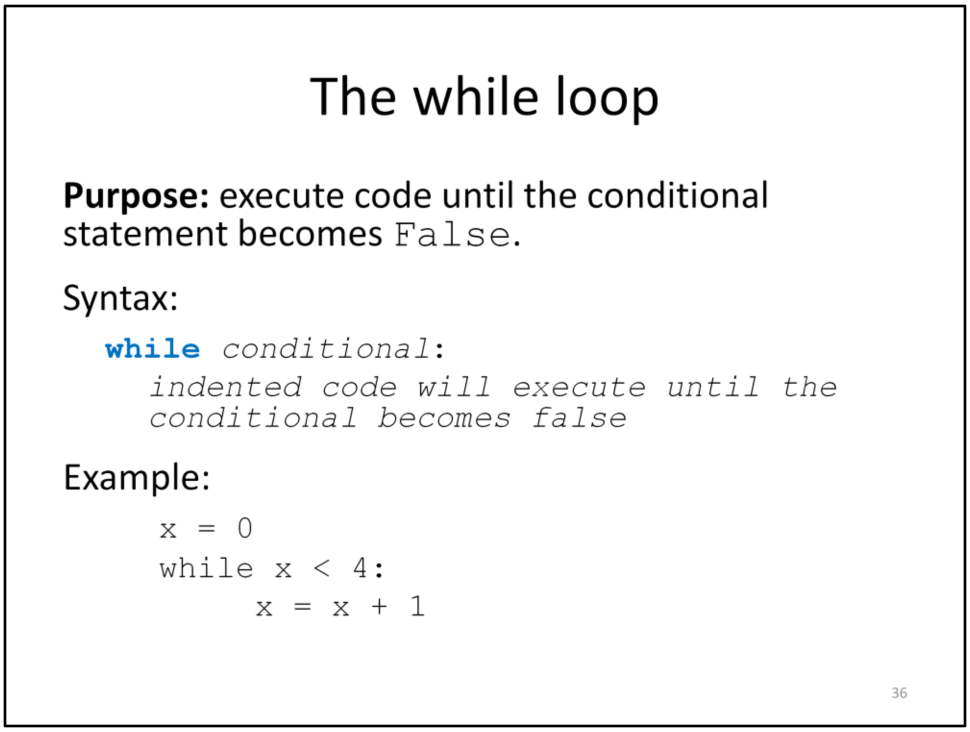
```
print random.randint(0,1)
print random.randint(0,1)
print random.randint(0,1)
print random.randint(0,1)
print random.randint(0,1)
print random.randint(0,1)
print random.randint(0,1)
print random.randint(0,1)
print random.randint(0,1)
print random.randint(0,1)
```

You can write:

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

Or :

```
count = 0
while count < 10:
    print random.randint(0,1)
    count = count + 1
```



The while loop

Purpose: execute code until the conditional statement becomes `False`.

Syntax:

```
while conditional:
  indented code will execute until the
  conditional becomes false
```

Example:

```
x = 0
while x < 4:
     x = x + 1
```

36

How is this different than a for loop?
Unlike the for loop, you don't need to specify exactly how many times the loop will run.

Again, the indentation is what defines what is part of the loop

Practice with `while`

What will the following code print?

```
x = 0
while x < 4:
    print "hi"
    x = x + 1
```

Practice with `while`

What will the following code print?

```
x = 0
while x < 4:
    print "hi"
    x = x + 1
```

Result:

```
hi
hi
hi
hi
```

Practice with `while`

What will the following code print?

```
x = 0
while x < 4:
    print x
    x = x + 1
```

Practice with `while`

What will the following code print?

```
x = 0
while x < 4:
    print x
    x = x + 1
```

Result:

```
0
1
2
3
```

Practice with `while`

What will the following code print?

```
x = 0
while x < 4:
    x = x + 1
    print x
```

Practice with `while`

What will the following code print?

```
x = 0
while x < 4:
    x = x + 1
    print x
```

Result:

```
1
2
3
4
```

Practice with `while`

What will the following code print?

```
x = 0
while x < 4:
    x = x + 1
print x
```

Practice with `while`

What will the following code print?

```
x = 0
while x < 4:
    x = x + 1
print x
```

Result:

```
4
```

A more useful example:
Number guessing game

```
secretNumber = 56
notGuessed = True

while (notGuessed):
   guess = int(raw_input("What number am I thinking of? "))
   if (guess == secretNumber):
        print "Wow, you got it!"
        notGuessed = False
   else:
        print "Wrong, guess again."
```

A more useful example: Number guessing game

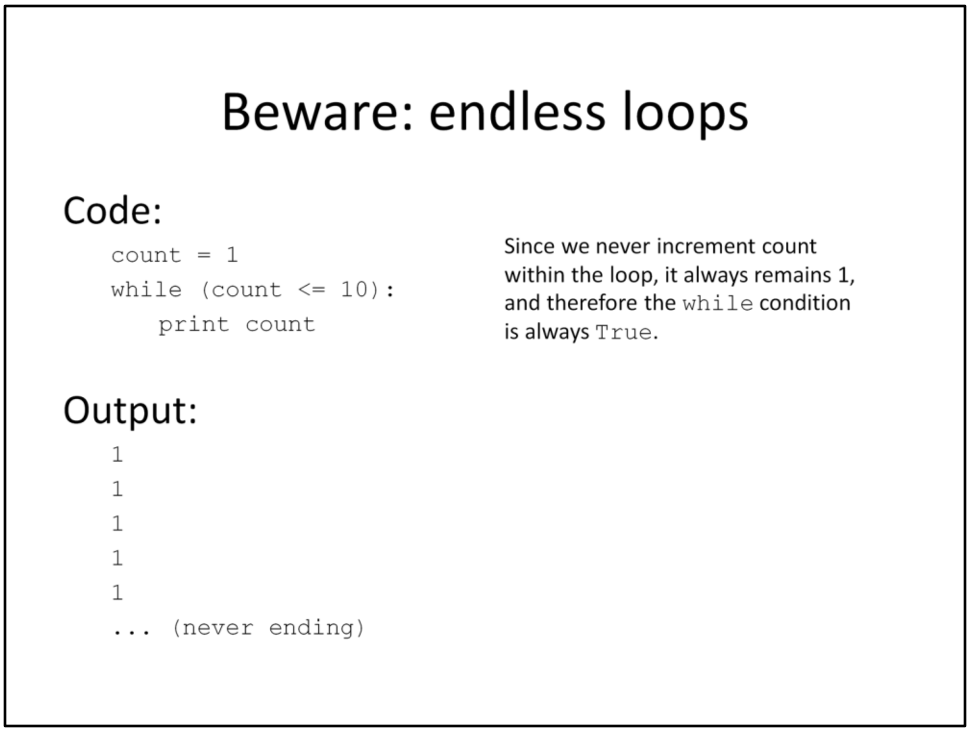
```
secretNumber = 56
notGuessed = True

while (notGuessed):
   guess = int(raw_input("What number am I thinking of? "))
   if (guess == secretNumber):
       print "Wow, you got it!"
       notGuessed = False
   else:
       print "Wrong, guess again."
```

this is initially `True`, so we enter the loop...

if the user guesses correctly, we simply set `notGuessed` to `False`. This makes the while loop condition `False`, and we therefore exit the loop.

if the user guesses wrong, we leave `notGuessed` as `True`, and therefore repeat the loop.

By using a `while` loop, we give the user unlimited chances to guess.



Beware: endless loops

Code:

```
count = 1
while (count <= 10):
   print count
```

Since we never increment count within the loop, it always remains 1, and therefore the `while` condition is always `True`.

Output:

```
1
1
1
1
1
... (never ending)
```

With while loops, you must be a little more careful than with for loops because of the possibility of creating an endless loop. An endless loop occurs when your conditional never becomes False, thus causing you to never exit the loop.

Endless loops

Always watch out for possible endless loops! If you're not sure, temporarily add a print statement somewhere in the loop so you can monitor how many times the loop runs.

If you find your code is taking an unexpectedly long time to run, check for an endless loop.

Stopping a program that is stuck in an endless loop:
Ctrl + c

More practice with `while` loops

Endless loop or not?

```
count = 0
while (count < 10):
    print count
    count = count + 1
```

More practice with `while` loops

Endless loop or not?

```
count = 0
while (count < 10):
   print count
   count = count + 1
```

Answer: **no**

More practice with `while` loops

Endless loop or not?

```
count = 0
while (count > 5):
   print count
   count = count + 1
```

More practice with `while` loops

Endless loop or not?

```
count = 0
while (count > 5):
    print count
    count = count + 1
```

Answer: **no**

(this won't print anything, actually, since the condition `count > 5` is never `True`)

More practice with `while` loops

Endless loop or not?

```
count = 0
while (count != 5):
   print count
   count = count + 1
```

More practice with `while` loops

Endless loop or not?

```
count = 0
while (count != 5):
   print count
   count = count + 1
```

Answer: **no**

More practice with `while` loops

Endless loop or not?

```
count = 0
while (count != 5):
   print count
count = count + 1
```

More practice with `while` loops

Endless loop or not?

```
count = 0
while (count != 5):
   print count
count = count + 1
```

Answer: **yes**

Why? We never increment `count` *within* the loop, so it never becomes equal to 5.

More practice with `while` loops

Endless loop or not?

```
count = 0
while (count != 5):
   print count
   count = count + 2
```

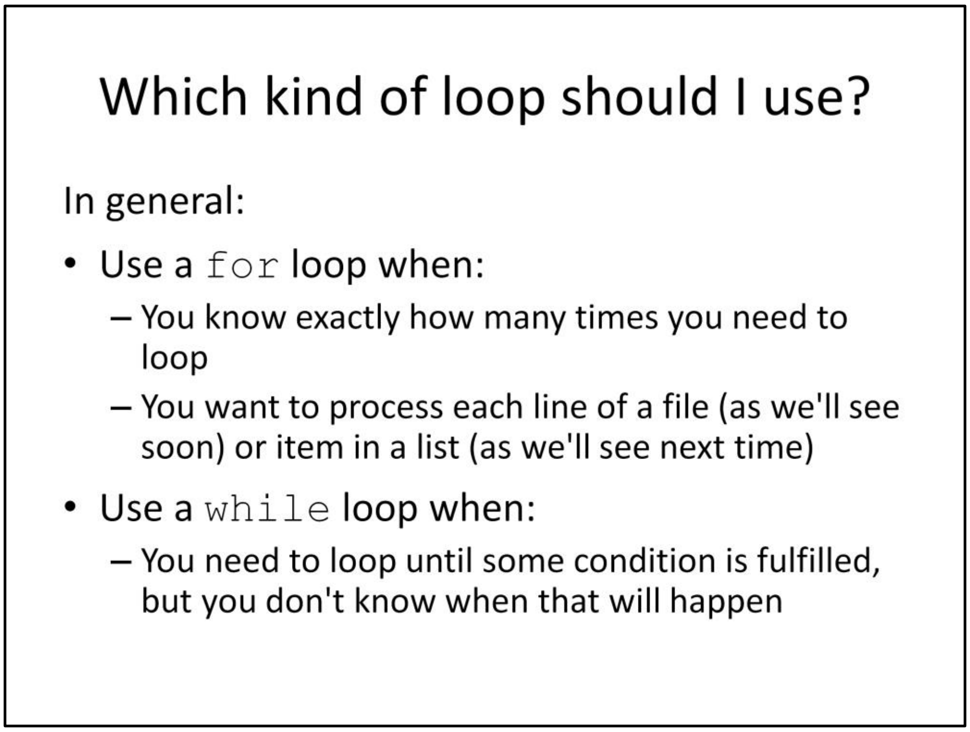
More practice with `while` loops

Endless loop or not?

```
count = 0
while (count != 5):
    print count
    count = count + 2
```

Answer: **yes**

Why? Since we're incrementing count by 2 each time, `count` takes the values 0, 2, 4, 6, 8, etc. `count` never equals 5, so the condition `count != 5` never becomes `False`, and we keep looping forever.



Which kind of loop should I use?

In general:

- Use a `for` loop when:
  - You know exactly how many times you need to loop
  - You want to process each line of a file (as we'll see soon) or item in a list (as we'll see next time)
- Use a `while` loop when:
  - You need to loop until some condition is fulfilled, but you don't know when that will happen

You can usually use either type of loop, but one will feel a lot more natural and be easier to code. That's the one you should go with.

4. Application of loops: file reading

File reading

- File reading (and writing) is something you'll probably be doing **a lot** in your work
- Luckily, Python makes it super easy!
- Today we'll cover file reading

File reading

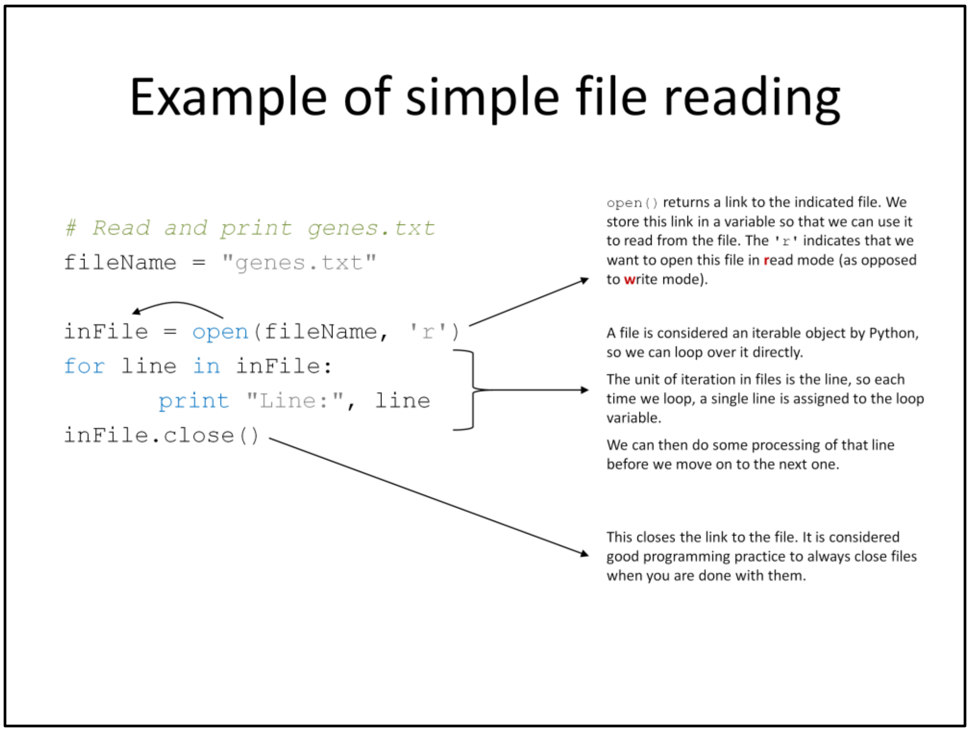
The 3 basic steps of file **reading**:

1. Open the input file
2. Read in data line by line, do some processing
3. Close the input file

File **writing** is very similar, but we'll save it for the next lesson.

Example of simple file reading

```
# Read and print genes.txt
fileName = "genes.txt"

inFile = open(fileName, 'r')
for line in inFile:
	print "Line:", line
inFile.close()
```



Example of simple file reading

```
# Read and print genes.txt
fileName = "genes.txt"

inFile = open(fileName, 'r')
for line in inFile:
      print "Line:", line
inFile.close()
```

`open()` returns a link to the indicated file. We store this link in a variable so that we can use it to read from the file. The `'r'` indicates that we want to open this file in **r**ead mode (as opposed to **w**rite mode).

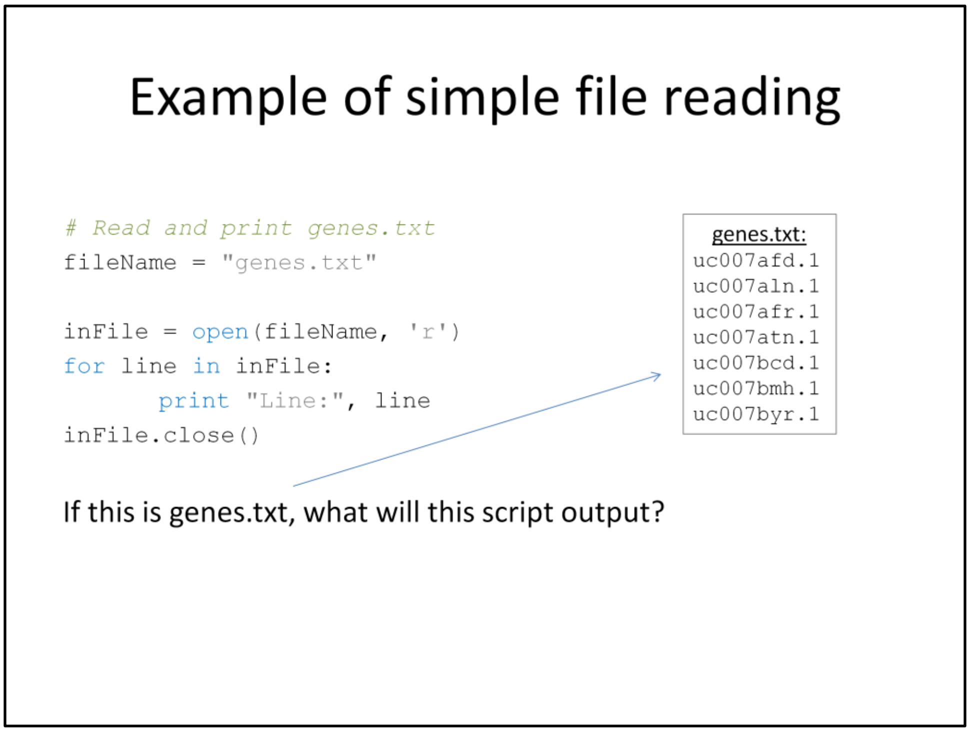
A file is considered an iterable object by Python, so we can loop over it directly.

The unit of iteration in files is the line, so each time we loop, a single line is assigned to the loop variable.

We can then do some processing of that line before we move on to the next one.

This closes the link to the file. It is considered good programming practice to always close files when you are done with them.

What open() technically does is create something called a File object, and this is what we store in the variable "inFile".
This file object is an iterable object, so we can use a loop on it. The individual iterable units are lines in the file, so during each iteration a single line is assigned to the loop variable. The lines will always be read from first to last.



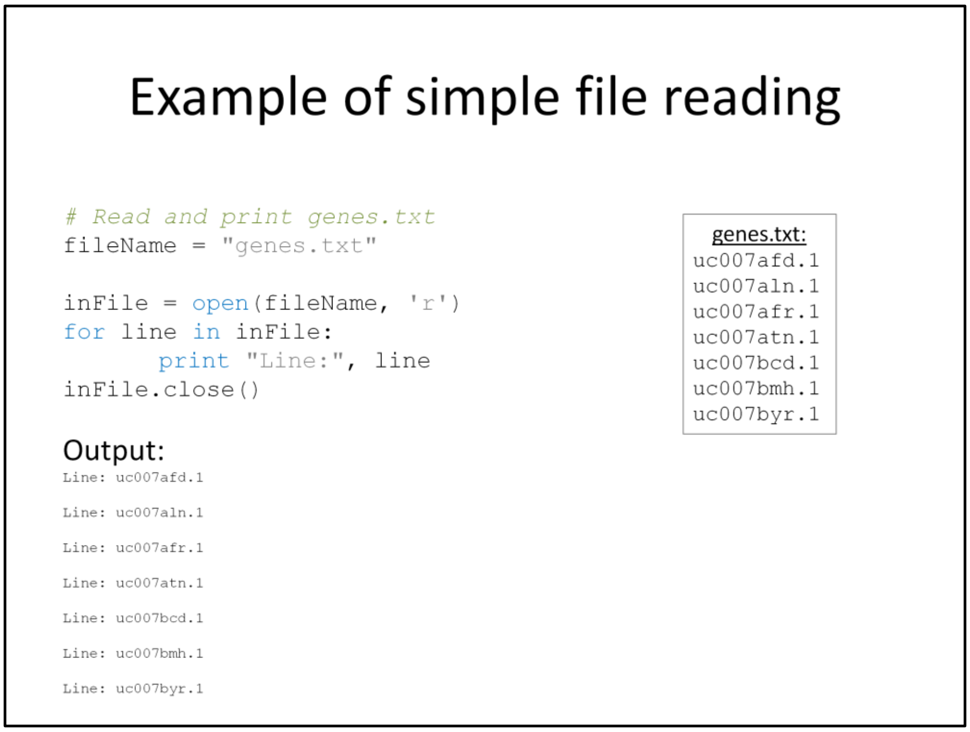
Example of simple file reading

```
# Read and print genes.txt
fileName = "genes.txt"

inFile = open(fileName, 'r')
for line in inFile:
        print "Line:", line
inFile.close()
```

<u>genes.txt:</u>

```
uc007afd.1
uc007aln.1
uc007afr.1
uc007atn.1
uc007bcd.1
uc007bmh.1
uc007byr.1
```

If this is genes.txt, what will this script output?



Example of simple file reading

```
# Read and print genes.txt
fileName = "genes.txt"

inFile = open(fileName, 'r')
for line in inFile:
        print "Line:", line
inFile.close()
```

<u>genes.txt:</u>

```
uc007afd.1
uc007aln.1
uc007afr.1
uc007atn.1
uc007bcd.1
uc007bmh.1
uc007byr.1
```

Output:

```
Line: uc007afd.1

Line: uc007aln.1

Line: uc007afr.1

Line: uc007atn.1

Line: uc007bcd.1

Line: uc007bmh.1

Line: uc007byr.1
```

Example of simple file reading

```
# Read and print genes.txt
fileName = "genes.txt"

inFile = open(fileName, 'r')
for line in inFile:
        print "Line:", line
inFile.close()
```

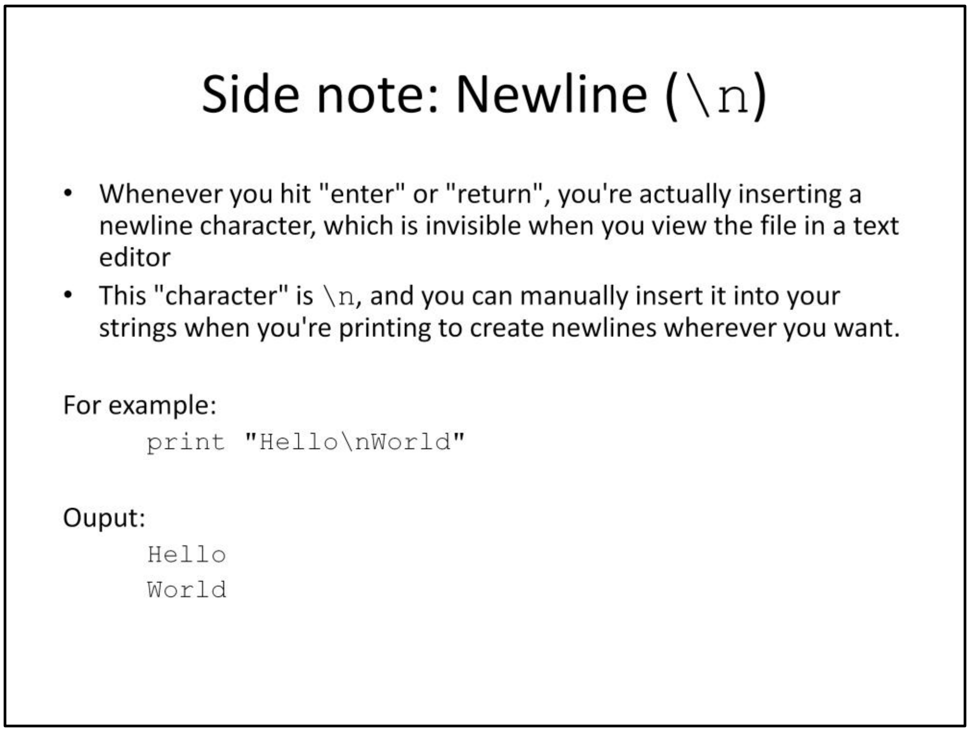
genes.txt:

```
uc007afd.1
uc007aln.1
uc007afr.1
uc007atn.1
uc007bcd.1
uc007bmh.1
uc007byr.1
```

Output:

```
Line: uc007afd.1

Line: uc007aln.1

Line: uc007afr.1

Line: uc007atn.1

Line: uc007bcd.1

Line: uc007bmh.1

Line: uc007byr.1
```

Why are there extra spaces?
Because of invisible `\n` characters!
When we read each line of the file, there is actually a `\n` on the end of each line. This gets read in as part of the string. Then `print` adds another `\n` on the end when it prints the string (as it always does). This is what causes the double spacing – we technically have `\n\n` on the end of each string.



Side note: Newline (`\n`)

- Whenever you hit "enter" or "return", you're actually inserting a newline character, which is invisible when you view the file in a text editor
- This "character" is `\n`, and you can manually insert it into your strings when you're printing to create newlines wherever you want.

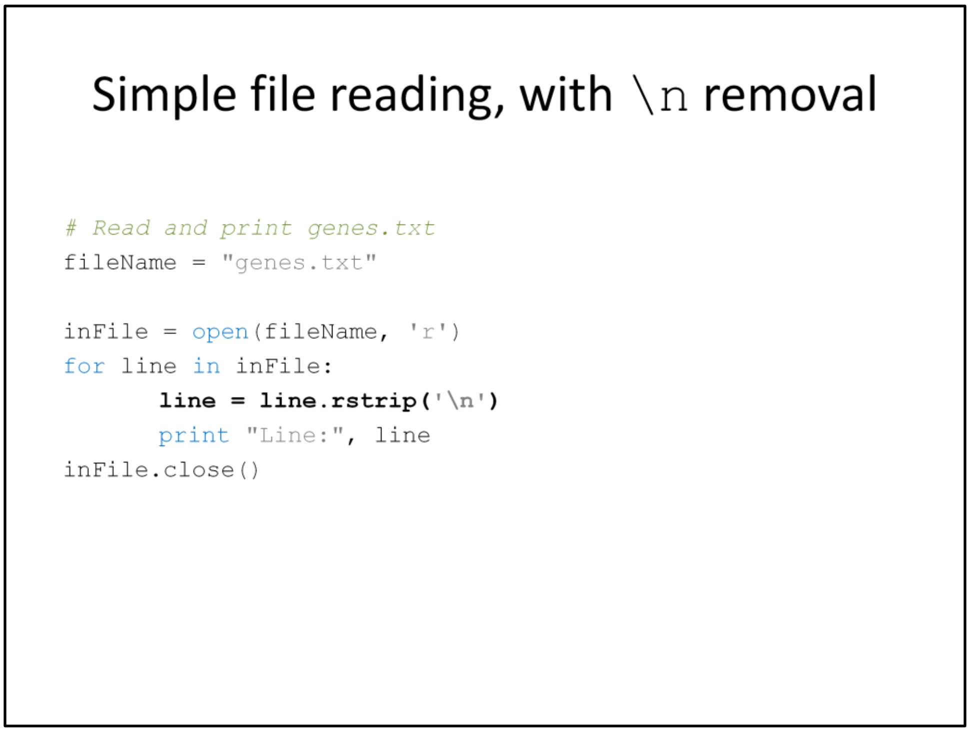
For example:

```
print "Hello\nWorld"
```

Ouput:

```
Hello
World
```

As you may have noticed, the print command automatically inserts a \n after it prints everything (if it didn't, you'd have everything running together on the same line when you use multiple print statements).



Simple file reading, with \n removal

```
# Read and print genes.txt
fileName = "genes.txt"

inFile = open(fileName, 'r')
for line in inFile:
        line = line.rstrip('\n')
        print "Line:", line
inFile.close()
```

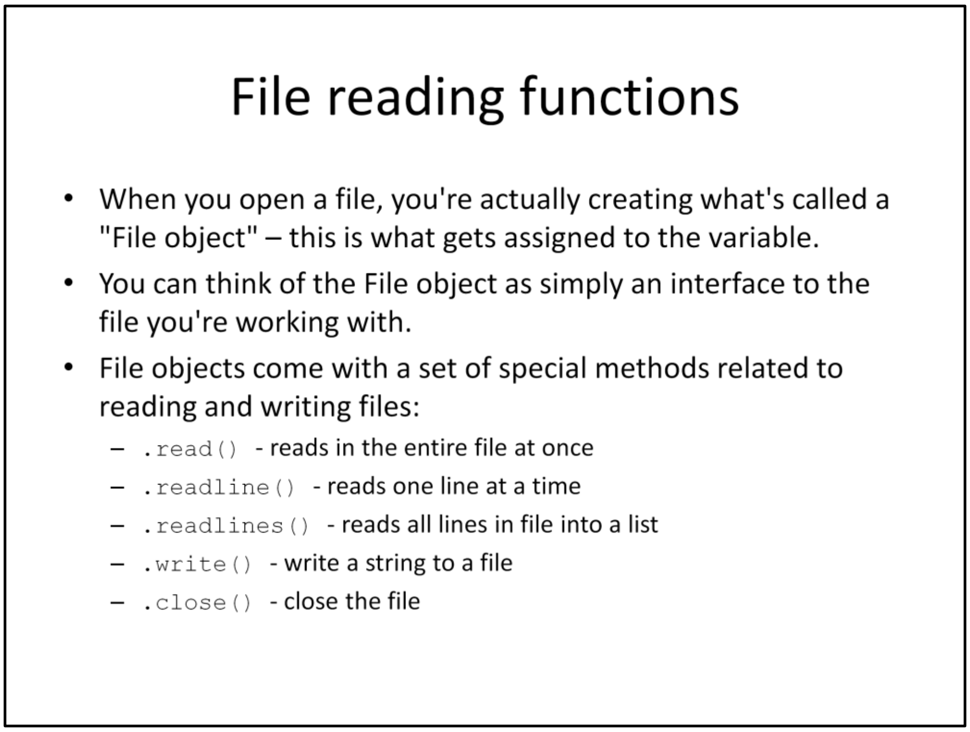
You can pretty much use this as a template for most file reading situations. The main part you'll be changing is to replace
  print "Line:", line
with something more interesting/useful

Simple file reading, with `\n` removal

```
# Read and print genes.txt
fileName = "genes.txt"

inFile = open(fileName, 'r')
for line in inFile:
        line = line.rstrip('\n')
        print "Line:", line
inFile.close()
```

`.rstrip()` removes the indicated character from the **end** of the string, if it is there. If the indicated character is not there, does nothing.

There are many cases when the `\n` will interfere with what you want to do, so it's good to get in the habit of including this line of code.



File reading functions

- When you open a file, you're actually creating what's called a "File object" – this is what gets assigned to the variable.
- You can think of the File object as simply an interface to the file you're working with.
- File objects come with a set of special methods related to reading and writing files:
  - `.read()` - reads in the entire file at once
  - `.readline()` - reads one line at a time
  - `.readlines()` - reads all lines in file into a list
  - `.write()` - write a string to a file
  - `.close()` - close the file

These methods can only be used on file objects. The syntax for using them is similar to what we've seen with modules, e.g.:

```
inFile.read()
inFile.readline()
```

Where "inFile" should be replaced with whatever variable name you assigned to your file when you opened it.

File reading functions

Examples:

```
inFile = open("genes.txt", 'r')  #create file object

header = inFile.readline()       #read first line of file
line = inFile.readline()         #read second line of file
restOfLines = inFile.readlines() #read rest into list

inFile.close()                   #clean up after ourselves
```