Iterations, Loops, String Operations
Program:

```python
n = 5
while n > 0 :
    print n
    n = n - 1
print 'Blastoff!'
print n
```

Output:

```
5
4
3
2
1
Blastoff!
0
```

Loops (repeated steps) have iteration variables that change each time through a loop. Often these iteration variables go through a sequence of numbers.
A Simple While Statement

n=5
while n>0:
    print n
    n=n-1
print 'Blastoff!'
print n

• More formally, here is the flow of execution for a while statement:

1. Evaluate the condition, yielding True or False.

2. If the condition is False, exit the while statement and continue execution at the next statement.

3. If the condition is True, execute the body and then go back to step 1.
An Infinite Loop

```
n = 5
while n > 0 :  
    print 'Lather'
    print 'Rinse'
print 'Dry off!'
```

What is wrong with this loop?
Another Loop

```python
n = 0
while n > 0 :
    print 'Lather'
    print 'Rinse'
    print 'Dry off!'
```

What does this loop do?
• Sometimes it’s more convenient to determine when to exit the while loop in its body, as opposed to by checking a condition

• For that case, we have the `break` statement
Breaking Out of a Loop

- The `break` statement ends the current loop and jumps to the statement immediately following the loop.

- It is like a loop test that can happen anywhere in the body of the loop.

```python
while True:
    line = raw_input('> ')
    if line == 'done':
        break
    print line
print 'Done!'  
```

```plaintext
> hello there
  hello there
> finished
  finished
> done
  Done!
```
Breaking Out of a Loop

• The **break** statement ends the current loop and jumps to the statement immediately following the loop

• It is like a loop test that can happen anywhere in the body of the loop

```python
while True:
    line = raw_input('> ')
    if line == 'done':
        break
    print line
print 'Done!'
```

> hello there
hello there
> finished
finished
> done
Done!
Finishing an Iteration with continue

The `continue` statement ends the current iteration and jumps to the top of the loop and starts the next iteration.

```python
while True:
    line = raw_input('> ')
    if line[0] == '#':
        continue
    if line == 'done':
        break
print line
print 'Done!'`
Wait!

- What does this condition mean?
  
  \[ \text{line}[0] == \text{‘#’} \]

- A line of text is treated in some respects like a list of characters, each of which can be referred to by its index

- This is true of all strings
  
  \[
  \text{myName} = \text{‘Yigal Arens’}
  
  \text{myName}[5] == \text{<what?>}
  \]

- But assignment, e.g., \text{myName}[5] = ‘x’ is NOT legal
Finishing an Iteration with continue

The `continue` statement ends the *current iteration* and jumps to the top of the loop and starts the next iteration.

```python
while True:
    line = raw_input('>' )
    if line[0] == '#':
        continue
    if line == 'done':
        break
    print line
print 'Done!'  >>> hello there
hello there
>>> # don't print this
>>> print this!
print this!
>>> done
Done!
```
while True:
    line = raw_input('> ')
    if line[0] == '#':
        continue
    if line == 'done':
        break
    print line
print 'Done!'
Indefinite Loop

• A while loop is an *indefinite loop*, because it keeps on iterating an indefinite number of times, until its logical condition becomes False.

• Sometimes we want a loop to iterate a specific number of times
Definite Loops

• Quite often we have a list of items of the lines in a file - effectively a finite set of things

• We can write a loop to run the loop once for each of the items in a set using the Python for construct

• These loops are called “definite loops” because they execute an exact number of times

• We say that “definite loops iterate through the members of a set”
A Simple Definite Loop

```
for i in [5, 4, 3, 2, 1] :
    print i
print 'Blastoff!'  
```

```
5
4
3
2
1
Blastoff!
```
A Definite Loop with Strings

```python
friends = ['Joseph', 'Glenn', 'Sally']
for friend in friends:
    print('Happy New Year:', friend)
print('Done!'
```

- Happy New Year: Joseph
- Happy New Year: Glenn
- Happy New Year: Sally
- Done!
Let’s Look More Carefully

A Simple Definite Loop

```
for i in [5, 4, 3, 2, 1] :
    print i
    print 'Blastoff!'  
```

Definite loops (for loops) have explicit iteration variables that change each time through a loop. These iteration variables move through the sequence or set.
Looking at In...

- The **iteration variable** “iterates” through the **sequence** (ordered set)
- The **block (body)** of code is executed once for each value in the **sequence**
- The **iteration variable** moves through all of the values in the **sequence**

```
for i in [5, 4, 3, 2, 1] :
    print i
```
- The iteration variable “iterates” through the sequence (ordered set)
- The block (body) of code is executed once for each value in the sequence
- The iteration variable moves through all of the values in the sequence

```python
for i in [5, 4, 3, 2, 1] :
    print i
```
for i in [5, 4, 3, 2, 1] :
    print i
Making “smart” loops

The trick is “knowing” something about the whole loop when you are stuck writing code that only sees one entry at a time.

- Set some variables to initial values
- for thing in data:
  - Look for something or do something to each entry separately, updating a variable
  - Look at the variables
Example: Find Largest Number in a List Containing Positive Numbers

numbers = [14, 3, 2, 25, 8, 10]

• How would you find the largest number?

• Remember:
  • Think of some variable to set initially
  • Go through the numbers, changing that variable if necessary
  • Check the variable’s final result

• What would that variable be?
Which Is the Largest Number?

14   3   2   25   8   10
Example: Find Largest Number in a List Containing Positive Numbers

```
numbers = [14, 3, 2, 25, 8, 10]
largestSoFar = -1
for num in numbers:
    if num > largestSoFar:
        largestSoFar = num
print 'Largest number was', largestSoFar
```
Example: Find Average of Numbers in a List

numbers = [14, 3, 2, 25, 8, 10]

• How would you find the average?

• Remember:
  • Think of some variable(s) to set initially
  • Go through the numbers, changing that variable(s) if necessary
  • Check the variable(s) final result

• What would those variables be?
Example: Find Average of Numbers in a List

```python
count = 0
sumSoFar = 0
for num in numbers:
    count = count + 1
    sumSoFar = sumSoFar + num
average = float(sumSoFar) / count
print 'There were', count, 'numbers'
print 'Their average was', average
```
String Operations
Strings Recap

- A string literal uses double or single quotes
  
  ```python
  >>> string1 = 'Bob'
  ```

- The operator `+` on strings concatenates them
  
  ```python
  >>> print string1 + '12' + 'Alice'
  Bob12Alice
  ```

- A string that contains numbers is still a string

- Can convert the **right** string to a number with `int()` and `float()`
  
  ```python
  >>> string2 = '1.5'
  >>> string3 = '100'
  >>> float(string2)
  1.5
  >>> int(string3)
  100
  >>> float(string3)
  100.0
  ```
Strings Recap

• We prefer to read input as a string, then parse it and convert it as needed

• This gives us more control over bad user input

• But raw input numbers must be converted!

  >>> num = raw_input(‘Enter number: ‘)
  Enter number: 13
  >>> print int(num) / 2
  6

• What would have happened had I not used int?
Looking Inside Strings

• We can look at any character in a string using an index in square brackets, e.g., `str[5]`
  
  >>> myName = ‘Yigal Arens’
  >>> print myName[2]
  <what do I get?>

• The index can be computed
  
  >>> x = 2
  >>> print myName[x ** 2]
  <what do I get?>

• What happens if I try the following?
  
  >>> print myName[x ** 5]
  <what do I get?>
Strings Have Length

- There is a built-in function `len` that gives us the length of a string

```python
>>> fruit = 'banana'
>>> print(len(fruit))
6
```
Len Function

```python
>>> fruit = 'banana'
>>> x = len(fruit)
>>> print x
6
```

A function is some stored code that we use. A function takes some input and produces an output.

Guido wrote this code

What is `fruit[len(fruit)]`?
Looping Through Strings

• Using a `while` statement and an `iteration variable`, and the `len` function, we can construct a loop to look at each of the letters in a string individually

```python
fruit = 'banana'
index = 0
while index < len(fruit):
    letter = fruit[index]
    print(index, letter)
    index = index + 1
```

<table>
<thead>
<tr>
<th>0</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a</td>
</tr>
<tr>
<td>2</td>
<td>n</td>
</tr>
<tr>
<td>3</td>
<td>a</td>
</tr>
<tr>
<td>4</td>
<td>n</td>
</tr>
<tr>
<td>5</td>
<td>a</td>
</tr>
</tbody>
</table>
Looping Through Strings

• A definite loop using a `for` statement is much more elegant

• The iteration variable is completely taken care of by the `for` loop

```python
fruit = 'banana'
for letter in fruit:
    print(letter)

index = 0
while index < len(fruit):
    letter = fruit[index]
    print(letter)
    index = index + 1
```
Looking deeper into `in`

- The iteration variable “iterates” through the sequence (ordered set)
- The block (body) of code is executed once for each value in the sequence
- The iteration variable moves through all of the values in the sequence

```python
for letter in 'banana':
    print letter
```

Six-character string
• We can also look at any continuous section of a string using a colon operator.

• The second number is one beyond the end of the slice - “up to but not including”

• If the second number is beyond the end of the string, it stops at the end.

```python
>>> s = 'Monty Python'
>>> print s[0:4]
Mont
>>> print s[6:7]
P
>>> print s[6:20]
Python
```
Slicing Strings

- If we leave off the first number or the last number of the slice, it is assumed to be the beginning or end of the string respectively.

```python
>>> s = 'Monty Python'
>>> print s[:2]
Mo
>>> print s[8:]
thon
>>> print s[:]
Monty Python
```
Using **in** as a logical Operator

- The **in** keyword can also be used to check to see if one string is “in” another string

- The **in** expression is a logical expression that returns **True** or **False** and can be used in an **if** statement

```python
>>> fruit = 'banana'
>>> 'n' in fruit
True
>>> 'm' in fruit
False
>>> 'nan' in fruit
True
>>> if 'a' in fruit:
    ...    print 'Found it!'
...    print 'Found it!'
...    print 'Found it!'
```
String Comparison

```python
if word == 'banana':
    print 'All right, bananas.'

if word < 'banana':
    print 'Your word, ' + word + ', comes before banana.'
elif word > 'banana':
    print 'Your word, ' + word + ', comes after banana.'
else:
    print 'All right, bananas.'
```

Let's try it
Libraries of Functions/Methods

• There are *libraries* of functions (AKA “methods”) associated with certain types of objects in Python

• Some are built-in, and later we’ll learn how to define our own

• The string library has many useful functions

• We call them by appending the function to a string variable, with a ‘.’ between them

• (This is an example of a more general facility we’ll learn about later.)
String Library

- Python has a number of string functions which are in the string library.

- These functions are already built into every string - we invoke them by appending the function to the string variable.

- These functions do not modify the original string, instead they return a new string that has been altered.

```python
>>> greet = 'Hello Bob'
>>> zap = greet.lower()
>>> print zap
hello bob
>>> print greet
Hello Bob
>>> print 'Hi There'.lower()
hi there
>>> ```
String Methods

• capitalize, center, count, decode, encode, endswith, expandtabs, find, format, index, isalnum, isalpha, isdigit, islower, isspace, istitle, isupper, join, ljust, lower, lstrip, partition, replace, rfind, rindex, rjust, rpartition, rsplit, rstrip, split, splitlines, startswith, strip, swapcase, title, translate, upper, zfill

• Or execute the Python command >>> dir(‘string’)

• The string functions/methods are defined here:
  https://docs.python.org/2/library/stdtypes.html#string-methods

A Few Examples

str.capitalize()
Return a copy of the string with its first character capitalized and the rest lowercased.
For 8-bit strings, this method is locale-dependent.

str.center(width[, fillchar])
Return centered in a string of length width. Padding is done using the specified fillchar (default is a space).

str.count(sub[, start[, end]])
Return the number of non-overlapping occurrences of substring sub in the range [start, end]. Optional arguments start and end are interpreted as in slice notation.
Particularly Useful String Functions

str.capitalize()
str.center(width[, fillchar])
str.endswith(suffix[, start[, end]])
str.find(sub[, start[, end]])
str.lstrip([chars])
str.replace(old, new[, count])
str.lower()
str.rstrip([chars])
str.strip([chars])
str.upper()
How to Extract A Substring

• This is what a typical first line in an email looks like

From aren@isi.edu Mon Sep  5 15:24:10 2016

• Suppose I would like to extract the host the message was sent from

>>> line = 'From aren@isi.edu Mon Sep  5 15:24:10 2016'
Extracting A Substring

• This is what a typical first line in an email looks like
  From arens@isi.edu Mon Sep  5 15:24:10 2016

• Suppose I would like to extract the host the message was sent from

```python
>>> line = 'From arens@isi.edu Mon Sep  5 15:24:10 2016'
>>> atpos = line.find('@')
>>> sppos = line.find(' ', atpos)
>>> hostname = line[atpos + 1 : sppos]
```