More About `==` (Equality) and `is`

```python
>>> x = 2
>>> y = 2
>>> x is y
True

>>> x == y
True

>>> y = x
>>> x is y
True

>>> x[0] = 'a'
>>> y
['a', 2]
```

```python
>>> x = [1, 2]
>>> y = [1, 2]
>>> x is y
False

>>> x == y
True

>>> x = y
>>> x is y
True
```
Iterations, Loops, String Operations
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

n = 0

while n > 0:
    print('Lather')
    print('Rinse')
    print('Dry off!')

What is this loop doing?
• 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 = input('> ')
    if line == 'done':
        break
    print(line)
print('Done!')
```

> 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 = 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 = 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!
Wait!

- What does this condition mean?
  
  ```
  line[0] == '#'
  ```

- 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
  
  ```
  myName = 'Yigal Arens'
  ```

  ```
  myName[5] == <what?>
  ```

- But assignment, e.g., `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 = 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!
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

```python
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**

```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

\[ \text{numbers} = [14, 3, 2, 25, 8, 10] \]

• 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

```python
numbers = [14, 3, 2, 25, 8, 10]
largestSoFar = -1
for num in numbers:
    if num > largestSoFar:
        largestSoFar = num
print ('Largest number was', largestSoFar)
```

25
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

numbers = [14, 3, 2, 25, 8, 10]
count = 0
sumSoFar = 0
for num in numbers:
    count = count + 1
    sumSoFar = sumSoFar + num
average = sumSoFar / count
print (‘There were’, count, ‘numbers’)  
print (‘Their average was’, average)
Example: Search Using a Boolean Variable — Search for 3

numbers = [14, 3, 2, 25, 8, 10]
found = False
print (‘Before’, found)
for value in numbers:
    if value == 3:
        found = True
        print (‘Found’, value)
print (‘After’, found)

How could this be improved?
Finding the Smallest Value

```python
smallest = None
print('Before')
for value in [9, 41, 12, 3, 74, 15]:
    if smallest is None:
        smallest = value
    elif value < smallest:
        smallest = value
print(smallest, value)
print('After', smallest)
```

We still have a variable that is the `smallest` so far. The first time through the loop `smallest` is `None`, so we take the first `value` to be the `smallest`. 

$ python smallest.py
Before
9 9
9 41
9 12
3 3
3 74
3 15
After 3
Let’s Try Some Code
Summary

- While loops (indefinite)
- Infinite loops
- Using break
- Using continue
- None constants and variables

- For loops (definite)
- Iteration variables
- Loop idioms
- Largest or smallest
String Operations
Strings: Recap of What We Know

• A string literal uses double or single quotes
  >>> string1 = 'Bob'

• The operator + on strings concatenates them
  >>> 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()
  >>> string2 = '1.5'
  >>> string3 = '100'
  >>> float(string2)
  1.5
  >>> int(string3)
  100
  >>> float(string3)
  100.0
  >>> int(string2)
  ????
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 = input('Enter number: ')
Enter number: 13
>>> print (int(num) / 2)
6.5

• 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

The built-in function `len` gives us the length of a string.

```python
>>> fruit = 'banana'
>>> print(len(fruit))
6
```
Looping Through Strings

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

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

This is a simple loop that loops through each letter in a string and counts the number of times the loop encounters the 'a' character.

```python
word = 'banana'
count = 0
for letter in word:
    if letter == 'a':
        count = count + 1
print(count)
```
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)
```
### Slicing Strings

<table>
<thead>
<tr>
<th>Monty</th>
<th>Python</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
</tr>
</tbody>
</table>

- 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
```
String Concatenation

When the + operator is applied to strings, it means “concatenation”

```python
>>> a = 'Hello'
>>> b = a + 'There'
>>> print(b)
HelloThere
>>> c = a + ' ' + 'There'
>>> print(c)
Hello There
>>> 
```
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.

```
>>> 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!')
Found it!
>>> 
```
String Comparison

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.')
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.)
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/3/library/stdtypes.html#string-methods

<table>
<thead>
<tr>
<th>A Few Examples</th>
<th>str.capitalize()</th>
</tr>
</thead>
<tbody>
<tr>
<td>Return a copy of the string with its first character capitalized and the rest lowercased.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>str.center(width[, fillchar])</th>
</tr>
</thead>
<tbody>
<tr>
<td>Return centered in a string of length width. Padding is done using the specified fillchar (default is an ASCII space). The original string is returned if width is less than or equal to len(s).</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>str.count(sub[, start[, end]])</th>
</tr>
</thead>
<tbody>
<tr>
<td>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.</td>
<td></td>
</tr>
</tbody>
</table>
**str.replace(old, new[, count])**

Return a copy of the string with all occurrences of substring `old` replaced by `new`. If the optional argument `count` is given, only the first `count` occurrences are replaced.

**str.rfind(sub[, start[, end]])**

Return the highest index in the string where substring `sub` is found, such that `sub` is contained within `s[start:end]`. Optional arguments `start` and `end` are interpreted as in slice notation. Return -1 on failure.

**str.rindex(sub[, start[, end]])**

Like `rfind()` but raises `ValueError` when the substring `sub` is not found.

**str.rjust(width[, fillchar])**

Return the string right justified in a string of length `width`. Padding is done using the specified `fillchar` (default is an ASCII space). The original string is returned if `width` is less than or equal to `len(s)`.

**str.rpartition(sep)**

Split the string at the last occurrence of `sep`, and return a 3-tuple containing the part before the separator, the separator itself, and the part after the separator. If the separator is not found, return a 3-tuple containing two empty strings, followed by the string itself.

**str.split(sep=None, maxsplit=-1)**

Return a list of the words in the string, using `sep` as the delimiter string. If `maxsplit` is given, at most `maxsplit` splits are done, the rightmost ones. If `sep` is not specified or `None`, any whitespace string is a separator. Except for splitting from the right, `rsplit()` behaves like `split()` which is described in detail below.
Other 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()
Making everything **UPPER CASE**

- You can make a copy of a string in **lower case** or **upper case**
- Often when we are searching for a string using `find()` we first convert the string to lower case so we can search a string regardless of case

```python
>>> greet = 'Hello Bob'
>>> nnn = greet.upper()
>>> print(nnn)
HELLO BOB
>>> www = greet.lower()
>>> print(www)
hello bob
>>> 
```
Searching a String

- We use the `find()` function to search for a substring within another string.
  - `find()` finds the first occurrence of the substring.
  - If the substring is not found, `find()` returns -1.
  - Remember that string position starts at zero.

```python
>>> fruit = 'banana'
>>> pos = fruit.find('na')
>>> print(pos)
2
>>> aa = fruit.find('z')
>>> print(aa)
-1
```
The `replace()` function is like a "search and replace" operation in a word processor.

- It replaces all occurrences of the search string with the replacement string.

```python
>>> greet = 'Hello Bob'
>>> nstr = greet.replace('Bob', 'Jane')
>>> print(nstr)
Hello Jane
>>> nstr = greet.replace('o', 'X')
>>> print(nstr)
HellX BXb
>>> 
```
How to Extract A Substring

• This is what a typical first line in an email looks like
  From arens@isi.edu Tue Jan 22 15:24:10 2018

• Suppose I would like to extract the host the message was sent from
  >>> line = 'From arens@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 Tue Jan 22 15:24:10 2018

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

```python
>>> line = 'From arens@isi.edu Tue Jan 22 15:24:10 2018'
>>> atpos = line.find('@')
>>> sppos = line.find(' ', atpos)
>>> hostname = line[atpos + 1 : sppos]
```
Summary

- String type
- Read/Convert
- Indexing strings []
- Slicing strings [2:4]
- Looping through strings with for and while
- Concatenating strings with +

- String operations
- String library
- String comparisons
- Searching in strings
- Replacing text
- Stripping white space