1 Strings

Strings are a very important data type in all languages. In Python, strings may be quoted several ways:

```
outputfule = 'output.txt'
triplequotes = """woah!
  split lines"""
print(triplequotes)  # Split onto multiple lines; newline char embedded.
```

This also works:

```
triple_single_quotes = '''I am a string too.
I can span multiple lines!'''
print(triple_single_quotes)
```

Quotes in quotes In Python, we can quote strings with either single quotes (i.e. a ') or double quotes (i.e. a "). But sometimes we want to create a string that contains quotes. Naively throwing in a quote won't work, since the interpreter will think it signifies the end of the string. But if we mix and match quote-types, it can easily be accomplished! Strings quoted with single-quotes can themselves contains double-quote characters:

```
'Bob said, "it is hot out there today".'
```

Strings quoted with a double-qoute can contains single quote characters:

```
"Python, it's a wonderful language"
```

Or we can escape the quote with a backslash, i.e. a \:

```
'it\'s not Nick\'s birthday today'
```

```
"I don't always quote my strings, but when I do, I prefer \""
```

1.1 Strings vs. Numbers

```
a = 5
b = '5'
a + b
# TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

```
print("type(a): ", type(a))  # <class 'int'>
print("type(b): ", type(b))  # <class 'str'>
```

It is simple to convert between numbers and strings!

```
# convert int to a string
a = str(55)
print(a)
print(type(a))  # Now it's a string!
```

```
# convert string to a float
a = float("99.45")
print(a)
print(type(a))  # Python knows how to parse strings to floats.
```

1.2 Slicing

Python is zero indexed!

```
quote = """That's all folks!"""
# Indices: 0123456789...
# Reversed:       ...7654321
print(quote[2])      # Extract the third character 'a' (Python is zero indexed)
print(quote[7:10])   # Extract sequence of chars, zero indexed 7,8,9 (excluding 10).
print(quote[:4])     # Take everything up to (but excluding) the fifth character.
print(quote[7:])     # Start from the eighth char, go all the way to the end.
print(quote[:-7])    # Go up to (but excluding) 7th char from end.
```

One way to remember how slices work is to think of the indices as pointing between characters, with the left edge of the first character numbered `0`. Then the right edge of the last character of a string of `n` characters has index `n`, for example:

```
word = 'Python'
```

```
+---+---+---+---+---+---+
| P | y | t | h | o | n |
+---+---+---+---+---+---+
  0   1   2   3   4   5
 -6  -5  -4  -3  -2  -1
```

```
word[-2:]  # Prints 'on'.
```

1.3 Strings are immutable

We can access an individual character of a string:

```
a = 'hello'
a[0]  # Prints 'h'
```

We *cannot* change any part of a string:

```
a[0] = 'k'  # Error: 'str' does not support item assignment.
```

1.4 Concatenation

Concatenate (add together) strings with '+':

```
b = 'j' + a[1:]  # Variable `b` now contains string "jello".
```

Here, we've created a new string variable.

1.5 String functions / methods

```
name = 'Leland'
len(name)
name.lower()  # Converts all chars to lowercase.
name.upper()  # Converts all chars to uppercase.

# Find the index of the first instance of 'lan' in the string.
name.find('lan')        # Returns 2 indicating that's where the match starts.
# Search for 'lan' within name[1:4], which contains substring 'ela'.
name.find('lan', 1, 4)  # Returns -1 indicating no match.
```

There are many string methods.

1.6 String formatting

It is often important to create strings formatted from a combination of strings, numbers, and other data. In Python 3 this is best handled by the `format` string method. E.g.:

```
name = "Andreas"
course = "CME211"
# Prints: My name is Andreas. I am the instructor for CME211.
print("My name is {0}. I am the instructor for {1}.".format(name,course))
```

Format strings contain "replacement fields" surrounded by curly braces {}. Anything that is not contained in braces is considered literal text, which is copied unchanged to the output. If you need to include a brace character in the literal text, it can be escaped by doubling the braces: i.e. use {{ and }}. The number in the braces refers to the order of arguments passed to `format`. Numbers don't need to be specified if the sequence of braces has the same order as arguments:

```
program = 'CME'
number = 211
print("this course is: {}-{}".format(program,number))
```

Realize that string formatting is a good way to combine text and numeric data. It's also how we control the output of floating point numbers:

```
# Fixed point precision (always uses six significant decimal digits).
print("    {{:f}}: {:f}".format(42.42))  # Prints 42.40000
# General format (knows how to drop trailing zeros in decimals).
print("    {{:g}}: {:g}".format(42.42))  # Prints 42.42
# Exponent (scientific) notation.
print("    {{:e}}: {:e}".format(42.42))  # Prints 4.242000e+01

# We can also specify how many digits of precision we want.
print("  {{:.2e}}: {:.2e}".format(42.42))    # Prints 4.24e+01.

# Or we can specify the width of our output (excluding +/- signs).
print("{{: 8.2e}}: {: 8.2e}".format(42.42))  # Prints total of 8 chars: 4.24e+01
print("{{: 8.2e}}: {: 8.2e}".format(-1.0))   # Prints -1.00e+00
```

See the Python Format Mini-Language docs and more examples

2 Numeric operations in Python

The following operations are defined for numeric types (i.e. `int` and `float`):

- `x + y`: sum of `x` and `y`
- `x - y`: difference of `x` and `y`
- `x * y`: product of `x` and `y`
- `x / y`: quotient of `x` and `y`
- `x // y`: floored quotient of `x` and `y`
- `x % y`: remainder of `x` / `y`
- `-x`: `x` negated
- `+x`: `x` unchanged
- `abs(x)`: absolute value (i.e. magnitude)
- `int(x)`: `x` converted to integer
- `float(x)`: `x` converted to floating point
- `x ** y`: `x` to the power `y`

A complete list of operations and more detailed discussion can be found in the Python

documentation.

2.1 Complex numbers

Python has built-in complex numbers:

```
c = 3 + 6j
print(c)          # (3+6j)
print(type(c))    # <class 'complex'>
```

Access real and imaginary parts:

```
print(c.real)  # 3.0
print(c.imag)  # 6.0
```

The parts themselves have type `float`:

```
print(type(c.real))  # <class 'float'>
print(type(c.imag))  # <class 'float'>
```

2.2 Numeric conversions

"Widening" It is often useful to convert between integers and floating point numbers. Python fully supports mixed arithmetic: when a binary arithmetic operator (such as `+` or `*`) has operands of different numeric types, the operand with the "narrower" type is widened to that of the other, where integer is narrower than floating point. Comparisons between numbers of mixed type use the same rule. The constructors `int()` and `float()` can be used to produce numbers of a specific type.

Let's see some examples:

```
x = 1 + 2.0
print(x)          # 3.0
print(type(x))    # <class 'float'>
```

In this case the integer `1` is "widened" or converted to the floating point number `1.0` before the addition. It is possible to manually convert from `int` to `float`:

```
x = float(3)
print(x)          # 3.0
print(type(x))    # <class 'float'>
```

It is also possible to convert from `float` to `int`:

```
x = int(4.7)
print(x)          # Note we get decimal truncation. Prints 4.
print(type(x))    # <class 'int'>
```

Let's see what happens with negative numbers:

```
x = int(-8.9)
print(x)          # -8
print(type(x))    # <class 'int'>
```

The Python *`int()` constructor* rounds floating point numbers *towards* `0`.

2.3 Converting to and from strings

Python makes it very easy to convert numbers to and from strings. This is a useful feature when trying to read numbers from a text file. Let's see it in action:

```
my_num_str = "42"
print(type(my_num_str))             # <class 'str'>
my_num_int = int(my_num_str)
my_num_float = float(my_num_str)
print(my_num_int)                   # 42
print(type(my_num_int))             # <class 'int'>
print(my_num_float)                 # 42.0
print(type(my_num_float))           # <class 'float'>
```

It also easy to convert from a numeric type back to a string:

```
print("exam score:" + str(95) + "%")
```

An attempt to concatenate a string with a numeric type is an error:

```
print("exam score:" + 95 + "%")
```

It is better to use string formatting for this:

```
print("exam score: {}%".format(95))
```

3 Lists

Sequential containers store data items in a specified order. Think elements of a matematical vector, ordered sequence of namesof people that you want to invite to your birthday party. The most fundamental Python data type for this is called a `list`. Later in the course

we will learn about containers that are more appropriate (and faster) for numerical data that come from NumPy.

3.1 Creating lists

In Python, lists are created by putting things inside of square brackets (`[]`).

```
some_primes = [1,2,3,5,7,11]
```

Lists can hold objects of any type:

```
many_types = [1, 55.5, "Am I in a list?", True]
```

We can get the length of the list with the `len()` functions:

```
len(many_types)  # Prints 4, the number of elements in the list.
```

3.2 Accessing Elements

Elements of a list are accessed using square brackets after a variable.

```
myList = [5, 2.3, 'hello']
```

The first element of a list is at index `0`:

```
mylist[0]  # Prints 5.
myList[2]  # Prints 'hello'.
```

Attempting to access an element out of bounds will produce an error:

```
mylist[3]  # Remember, Python is zero-indexed!
```

We can index to `-1` to get the object at the very end of the list:

```
myList[-1]  # Get the first element from the end, i.e. 'hello'.
```

Likewise, we can index backwards using negative numbers:

```
myList[-3]  # Get the third elemenet from the end.
```

3.3 Slicing

A sub-list may be accessed using slice syntax. Let's start with the list:

```
many_types = [1, 55.5, "Am I in a list?", True, "the end"]
```

Let's look at a sub-list:

```
many_types[2:4]  # Still zero indexed elements 2 and 3 (excludes index 4)
```

The `[2:4]` is called a *slice* and returns a list with elements at indices 2 and 3 from the original list. It is easy to slice from an index to the end:

```
many_types[2:]  # Start with third element, go to end.
```

It is also easy to slice from the beginning to a specified index:

```
many_types[:3]  # Extract elements indexed at positions 0, 1, and 2.
```

3.4 Adding and multiplying

The `+` operator concatenates (or combines) lists:

```
myList = [5, 2.3, 'hello']
mySecondList = ['a', '3']
concatList = myList + mySecondList
print(concatList)  # Prints: [5, 2.3, 'hello', 'a', '3']
```

The `*` operator can be used to repeat lists:

```
myList = ['hello', 'world']
print(myList * 2)  # These two results...
print(2 * myList)  # ...both print the list 2x.
```

3.5 Lists are mutable

Lists are mutable, this means that individual elements can be changed.

```
# start with a list
my_list = ['a', 43, 1.234]
# assign a new value to index 0, i.e. replace char 'a' with int -3.
my_list[0] = -3
# inspect the list
```

```
print(my_list)
```

We can also assign to a slice

```
x = 2
my_list[1:3] = [x, 2.3]
print(my_list)
```

3.6 Copying a list

Let's attempt to copy a list referenced by variable `a` to another variable `b`:

```
a = ['a', 'b', 'c']
b = a      # attempt to copy a to b
b[1] = 1   # now we want to change an element in b
print(b)   # Here we see we did swap second element in b...
print(a)   # ...but oops! We also modified second element in a.
```

That's interesting! Modifying `b` caused `a` to change.

Let's look at this example in Python Tutor. How about a different technique?

```
a = ['a', 'b', 'c']
b = a        # first attempt to copy a to b
c = list(a) # use the list constructor
b[1] = 1     # now we want to change an element in b
print("a: ", a)  # list 'a' got modified (unintentionally).
print("b: ", b)  # list 'b' of course got modified (intentionally).
print("c: ", c)  # list 'c' got preserved, success!

# The 'id' function prints out the memory address of an object.
print("id(a): ", id(a))  # Note that list 'a'...
print("id(b): ", id(b))  # ...and list 'b' have same id.
print("id(c): ", id(c))  # List 'c' has its own id.
```

Again, let's try this example in Python Tutor. A list can be copied with `b = list(a)` or `b = a[:]`. The second option is a slice including all elements!

3.7 Python's data model

Variables in Python are actually a reference to an object in memory. Assignment with the `=` operator sets the variable to refer to an object. Here is a simple example:

```
a = [1,2,3,4]
b = a           # Create a reference to the object 'a'.
b[1] = 55       # Overwrite the second elem in the obj. referenced by 'a' and 'b'.
print(b)        # Notice of course that both variables...
```

```
print(a)        # ...contain the same data.
```

In this example, we assigned `a` to `b` via `b = a`. This did not copy the data, it only copied the reference to the list object in memory. Thus modifying the list through `b` also changes the data that you will see when accessing from `a`. You can inspect object ids in Python with:

```
print("id(a): ", id(a))    # These two variables...
print("id(b): ", id(b))    # ...both refer to the same object.
```

Those numbers refer to memory addresses.

3.8 Copying data (more generally)

The `copy` function in the `copy` module is a more generic way to copy a list:

```
import copy
a = [5,2,7,0,'abc']
b = copy.copy(a)
b[4] = 'xyz'
print(b)  # Variable b has its last element replaced.
print(a)  # Variabla a is unchanged, like we hoped.
```

Note that elements in a list are also references to memory location. For example if your list contains a list, this will happen when using `copy.copy()`:

```
a = [2, 'string', [1,2,3]]
b = copy.copy(a)
b[2][0] = 55
print(b)  # We modified the nested list...
print(a)  # ...which also affected the contents of variable 'a'.
```

Let's visualize it in Python Tutor. Here, the element for the sub-list `[55, 2, 3]` is actually a memory reference. So, when we copy the outer list, only references for the contained objects are copied. Thus in this case modifying the copy (`b`) modifies the original (`a`). Thus, we may need the function `copy.deepcopy()`:

```
a = [2, 'string', [1,2,3]]
b = copy.deepcopy(a)
b[2][0] = 99
print(b)  # Variable 'b' has its third element modified.
print(a)  # Here, we see that variable 'a' is unchanged.
```

3.9 Sorting lists

Sorting Python lists is very easy. Let's randomly shuffle a list and then sort it.

```
import random
my_list = list(range(10))
print(my_list)              # [0, 1, 2, ..., 8, 9]
random.shuffle(my_list)     # In-place shuffle.
print(my_list)              # Some random permutation of the above.
```

Note that the `random.shuffle()` function shuffles the list in-place. It doesn't create a new list. We can use the `sorted()` function to return a *new* sorted list:

```
sorted_list = sorted(my_list)
print("my_list:", my_list)
print("sorted_list:", sorted_list)
```

The list `sort()` method sorts the list in place:

```
my_list.sort()
print("my_list:", my_list)  # Elements are printed in order...
```

3.10 List operations

In the following summary, `s` is a list and `x` is an element.

- `x in s`: `True` if an item of `s` is equal to `x`, else `False`
- `x not in s`: `False` if an item of `s` is equal to `x`, else `True`
- `s + t`, where `t` is another *list*: the concatenation of `s` and `t`.
- `s * n` or `n * s`, where n is an `int`: equivalent to adding `s` to itself `n` times
- `s[i]`: ith item of `s`, origin `0`
- `s[i:j]`: slice of `s` from `i` to `j`
- `s[i:j:k]`: slice of s from `i` to `j` with step `k`
- `len(s)`: length of `s`
- `min(s)`: smallest item of `s`
- `max(s)`: largest item of `s`
- `s.index(x)`: index of the first occurrence of `x` in `s`
- `s.count(x)`: total number of occurrences of `x` in `s`
- `s[i] = x`: item `i` of `s` is replaced by `x`

- `s[i:j] = t`: slice of `s` from `i` to `j` is replaced by the contents of the `t`
- `del s[i:j]`: same as `s[i:j] = []`
- `s[i:j:k] = t`: the elements of `s[i:j:k]` are replaced by those of `t`
- `del s[i:j:k]`: removes the elements of `s[i:j:k]` from the list
- `s.append(x)`: appends `x` to the end of the sequence.
- `s.clear()`: removes all items from `s` (same as `del s[:]`)
- `s.copy()`: creates a shallow copy of `s` (same as `s[:]`)
- `s.extend(t) or s += t`: extends `s` with the contents of `t` (for the most part the same as `s[len(s):len(s)] = t`)
- `s *= n`: updates `s` with its contents repeated `n` times
- `s.insert(i, x)`: inserts `x` into `s` at the index given by `i` (same as `s[i:i] = [x]`)
- `s.pop([i])`: retrieves the item at `i` and also removes it from `s`
- `s.remove(x)`: remove the first item from `s` where `s[i] == x`
- `s.reverse()`: reverses the items of `s` in place
- `s.sort()`: sorts the items of `s` in place

Also have a look at `help(list)`.

3.11 Resources for Lists

- Python tutorial section on `list`
- Python reference section on sequences

4 Looping with `for` and `while`

Very often, one wants to repeat some action. This can be achieved with `for` and `while` statements.

4.1 `for` loops

A `for` loop is typically used when we want to repeat an action a given number of times.

```
for i in range(5):
    print(i**2, end=', ')
print() # print a new line at the end
```

Here, `for i in range(n):` will execute the loop body `n` times with `i = 0, 1, 2, ..., n - 1` in succession.

4.2 Note on Python syntax

Python uses syntatic indenting. This means that indenting code has a meaning in the programming language. In languages like C, C++, and Java, loop bodies are enclosed in braces, but good coding style suggests that statements in a loop or conditional body are indented:

```
for (int i = 0; i < 10; i++) {
  printf("i = %d\n", i);
}
```

Python takes this a step further and requires the indenting of loop and conditional bodies. We recommend that you use 4 spaces to indent python code (so does the python community]). Please tell your text editors to insert spaces instead of tab characters when you hit the tab key on the keyboard.

4.3 The `range()` function

The `range()` function can be used in a few different ways. We can convert a range object to a python list with the `list()` function:

```
# get range 0,...,6
print(list(range(7)))
# get range 4,...,10
print(list(range(4,11)))
# get range [4,16) with step of 3
print(list(range(4,16,3)))                    # Prints [4, 7, 10, 13]
```

See `help(range)` for more info.

Note that `range` differs in Python 2 and 3. In Python 2, `range()` returns a list. In Python 3, `range()` returns an object that produces a sequence of integers in the context of a `for` loop, which is more efficient, because memory for a new list need not be allocated.

4.4 `for` and lists

We can use a `for` loop to iterate over items in a list:

```
my_list = [1, 45.99, True, "str item", ["sub", "list"]]
for item in my_list:
    print(item)
```

It is often handy to get access to both the list item and index in a `for` loop. This can be achieved with the `enumerate()` function:

```
my_list = [1, 45.99, True, "str item", ["sub", "list"]]
for index, item in enumerate(my_list):
    print("{}: {}".format(index, item))
## 1: 45.99
## 2: True
## 3: str item
## 4: ['sub', 'list']
```

4.5 Example adding numbers

```
summation = 0
for n in range(1,101):
    summation += n
print(summation)  # Prints 5050 (easy to check with Euler's formula).
```

Interlude - Euler's Formula Some of you may not have seen Euler's formula: it tells us that the sum of integers $1, 2, ..., n$ is equal to $\frac{n(n+1)}{2}$. Why? To see this, simply add the vector of elements $(1, 2, \ldots, n)$ element-wise to same vector in reverse order; the resulting vector contains homogeneous elements:

```
   1   +   2 +   3 + ... + n-1 + n    <-- This is what we want to sum
+  n   + n-1 + n-2 + ... +   2 + 1    <-- These are the same elements in reverse.
---------------------------------
   n+1 + n+1 + n+1 + ... + n+1 + n+1  <-- n of these terms.
```

If we summed the elements on the last row, we would be double counting the elements, whence we divide by 2. In any case, the result is also achievable in Python via `sum`:

```
sum(range(1,101))
```

4.6 `while` loops

When we do not know how many iterations are needed, we can use `while`.

```
i = 2
while i < 100:      # loop body only execute if conditional statement is True
    print(i**2,end=", ")
    i = i**2
```

This prints out the elements $2^2 = 4, 4^2 = 16, 16^2 = 256$, and of course since $256 >= 100$ we terminate the while loop at that point.

4.7 Infinite loops

```
while True:
    print("hah!")
```

I won't include the output...but! if you want to play with it, use `ctrl-c` to interrupt the interpreter.

4.8 Nesting loops

A *nested loop* is a loop in the body of a loop.

```
for i in range(8):
    for j in range(i):
        print(j, end=' ')
    print()

# Prints the following to console:
# 0
# 0 1
# 0 1 2
# 0 1 2 3
# 0 1 2 3 4
# 0 1 2 3 4 5
# 0 1 2 3 4 5 6
```

4.9 `continue`

`continue` continues with the next iteration of the smallest enclosing loop:

```
for num in range(2, 10):
    if num % 2 == 0:
```

```
3           print("Found an even number:", num)
4           continue
5       print("Found an odd number:", num)
```

Here, `num in range(2,10)` sets up a loop where `num = 2, 3, ..., 9`.

4.10 `break`

The `break` statement allows us to jump out of the smallest enclosing `for` or `while` loop.

Finding prime numbers:

```
1  max_n = 10
2  for n in range(2, max_n):
3      for x in range(2, n):
4          if n % x == 0: # n divisible by x
5              print(n, 'equals', x, '*', n/x)
6              break
7      else: # executed if no break in for loop
8          # loop fell through without finding a factor
9          print(n, 'is a prime number')
10
11 # Prints:
12 # 2 is a prime number
13 # 3 is a prime number
14 # 4 equals 2 * 2.0
15 # 5 is a prime number
16 # 6 equals 2 * 3.0
17 # 7 is a prime number
18 # 8 equals 2 * 4.0
19 # 9 equals 3 * 3.0
```

4.11 A note on Python variables

It is bad practice to define a variable inside of a conditional or loop body and then reference it outside:

```
1  name = "Nick"
2  if name == "Nick":
3      age = 45 # newly created variable
4  print("Nick's age is {}".format(age))
```

Here is what happens when a variable is not created:

```
1  name = "Bob"
2  if name == "Nick":
3      id_number = 45 # also newly created variable
```

```
print("{}'s id number is {}".format(name, id_number))
# NameError: name 'id_number' is not defined
```

Good practice to define/initialize variables at the same level they will be used:

```
name = "Bob" age = 55 if name == "Nick":      age = 45   print("{}'s age is {}".forma
```

5 Python File IO (Input-Output)

Python makes it very easy to read and write files to disk.

Keep in mind that it is almost always better to use a Python module for specific formats. For example, use the `json` module for JSON files or the `csv` module for `.csv` files. Better yet, use Pandas for table-like data.

5.1 What is a file?

A *file* is a segment of data, typically associated with a filename, that exists in a computer's persistent storage. This means that the data remains when the computer is turned off.

There are two main kinds of files: *text* and *binary*.

Text files are typically easier for humans to read and write.

Binary files (images, music files, etc.) are more efficient in terms of storage.

Python scripts are text files and by convention have a `.py` extension. On unix systems we can dump a text file to the terminal with:

```
$ cat hello.py
# run me from the command line with
# $ python3 hello.py

print("hello sweet world!")
```

For fun, try dumping a binary file to the terminal with `$ cat /bin/ls`. What happens?

In Python it is very easy to open, read from, and write to a text file. Let's see how it works.

See Chapter 9 in **Learning Python** for information on accessing files with Python. The relevant information starts on page 282.

5.2 The file object

- Interaction with the file system is pretty straightforward in Python.
- Done using *file objects*
- We can instantiate a file object using `open` or `file`

5.3 Opening a file

```
f = open(filename, option)
```

- `filename`: path to file on disk
- `option`: mode to open file (passed as a string)
- `'r'`: read file
- `'w'`: write to file (overwrites existing file)
- `'a'`: append to file
- We need to close a file after we are done: `f.close()`

Open a file:

```
f = open("humpty-dumpty.txt","r")
```

We can test if the file is closed:

```
f.closed  # Returns a Boolean (in this case False).
```

We can close the file:

```
f.close()
f.closed   # Now returns True.
```

Closing a file flushes any buffered data to disk and frees up operating system resources. If using a file in this manner, it is important to close files. *We will take off points if you neglect to do this.*

5.4 `with open() as f:`

It is good practice to use the `with` keyword when dealing with file objects. This has the advantage that the file is properly closed after its suite finishes, even if an exception is raised on the way.

```
with open('humpty-dumpty.txt', 'r') as f:
    print(f.read())
f.closed  # Returns True, that's the benefit of `with open(...) as ...:`.
```

5.5 If a file does not exist

```
bad_file = open("no-file.txt","r")
# FileNotFoundError: [Errno 2] No such file or directory: 'no-file.txt'
```

5.6 Reading data from a file

File object methods:

- `read()`: Read entire file (or first `n` characters, if supplied)
- `readline()`: Reads a single line per call
- `readlines()`: Returns a list with lines (splits at newline)

5.7 `readline()`

Use the `readline()` method to read lines from a file:

```
f = open("humpty-dumpty.txt","r")
print(f.readline())
print(f.readline())
f.close()
```

5.8 `read()`

You can read an entire file at once with the `read()` method:

```
f = open("humpty-dumpty.txt","r")
poem = f.read()
```

```
print(poem)
f.close()
```

5.9 Iterate over lines

You can very easily iterate over lines in a file with:

```
f = open("humpty-dumpty.txt","r")
for line in f:
    print(line)
f.close()
```

5.10 An example with `with`

```
with open('humpty-dumpty.txt', 'r') as f:
    for i, line in enumerate(f):
        print("line {}: {}".format(i,line))
```

Note there will be extra lines between each line of text if you run this. You can do this by specifying the `end` keyword parameter for the `print` function to be an empty string (`""`): `print(line, end='')` or slicing `line` with `print(line[:-1])`.

5.11 Iterate over words!

The string `split` method partitions a string into a list based on a delimiter. Space is the default delimiter. The `strip` method removes leading and trailing whitespace from a string.

```
f = open("humpty-dumpty.txt","r")
for line in f:
    for word in line.split():
        # use strip() method to remove extra newline characters
        print(word.strip())
f.close()
```

5.12 Writing to file

Use the `write()` method to write to a file. Make sure to open the file in write mode with `'w'` as the second argument to `open()`.

```
name = "Python learner"
with open('hello.txt', 'w') as f:
```

```
    f.write("Hello, {}!\n".format(name))
```

```
cat hello.txt
```

5.13 More writing examples

Write elements of list to file:

```
xs = ["i", "am", 'a', 'fancy', 'list', 42]
with open('from_list.txt', 'w') as f:
    for x in xs:
        f.write('{}\n'.format(x))
```

To write multiple lines to a file at once, use the `writelines` method:

```
f = open("writelines.txt","w")
f.writelines(["a mighty fine day\n", \
              "ends with a great big party\n", \
              "thank you, its friday\n"])
f.close()
```

Note that the `write` and `writelines` methods will not insert newline characters. To get a new line, you must add `'\n'` to the strings.

5.14 Buffering

For efficiency, the `file` object will temporarily store data from `write` or `writelines` methods in memory before actually writing to disk. This is known as buffering. It turns out that writing larger chunks of data to disk in fewer transactions is more efficient than many transactions of small chunks. If you attempt to open a text file created by Python and not closed, you may not see the data. Calling the `close()` method flushes all data to disk.

```
f = open('foo.txt','w')
f.write("this is some sweet text\n")
```

(On my system `foo.txt` is empty at this point. Behavior may be different on your system.)

```
f.close()
```