

Today: accumulate patterns - counting and summing, int modulo, text files, standard output, print(), file-reading, crazycat example program

s.find() 2 Param Form

(last thing from lecture-10 .. use in section and later)

s.find() variant with 2 params: s.find(target, start_index) - start search at start_index vs. starting search at index 0. Returns -1 if not found, as usual. Use to search in the string starting at a particular index.

Suppose we have the string 'xx[abc['. How to find the second '[' which is at 6? By default, search starts at 0. Start the search at 1, just after the first bracket:

```
>>> s = 'xx[abc['
>>> s.find('[')           # find first [ (start at 0)
2
>>> s.find('[', 3)       # start search at 3
6
```

Accumulate Pattern

The functions today and the old double_char() function fit in this rough pattern:

1. at the start: result = empty
at the end: return result
2. In the loop, some sort of: result += xxx

Working on a problem in this pattern, you have a head start on parts of it, and can concentrate on what parts are different.

Loop Counting

A common problem in computer code is **counting** the number of times something happens within a data set. This is within the pattern, using **count = 0** before the loop and **count += 1** in the loop. Recall that the line count += 1 will increase the int stored in the variable by 1.

```
count = 0

loop:
    if thing-to-count:
        count += 1

return count
```

Example count_e()

This string problem shows how to use `+= 1` to count the occurrences of something, in this case the number of 'e' in a string.

> [count_e\(\)](#)

count_e() Solution

```
def count_e(s):
    count = 0
    for i in range(len(s)):
        if s[i] == 'e':
            count += 1
    return count
```

Loop Summing

A related code problem is - how sum up a series of numbers? It's really the same pattern again. Set `sum = 0` before the loop. Inside the loop, use `sum += xxx` to add the desired value to the sum

```
sum = 0

loop:
    sum += one_number

return sum
```

Example shout_score()

> [shout_score\(\)](#)

Say we want to rate an email about how long and how much shouting it has in it before we read - like scoring emails from your nutty relatives.

shout_score(s): Given a string s, we'll say the "shout" score is defined this way: each exclamation mark '!' is 10 points, each lowercase char is 1 point, and each uppercase char is 2 points. Return the total of all the points for the chars in s.

'Arg!!' -> 24 points

'A' -> 2

'r' -> 1

'g' -> 1

'!' -> 10

'!' -> 10

In the loop, use the sum pattern to compute the score for the string.

shout_score() Solution

```
def shout_score(s):
    score = 0
    for i in range(len(s)):
        if s[i] == '!':
            score += 10
        elif s[i].islower():
            score += 1
        elif s[i].isupper():
            score += 2
    return score
```

Here using if/elif structure, since our intention is to pick out 1 of N tests. As a practical matter, it also works as a series of plain if. Since '!' and lowercase and uppercase chars are all exclusive from each other, only one if test will be true for each char.

Python - Behind the Scenes

Most often, what Python code will do follows your intuition. Here we'll look at the mechanism underlying how it works, and sometimes that will be important so you can understand how to make the code work.

Types - `int` and `str`

Q: What is the difference between these two?

`123` vs. `'123'`

These two values are different **types**. The type of a value is its official category, and all data in Python has a type. The formal type of integers is **`int`** and type of strings is **`str`**.

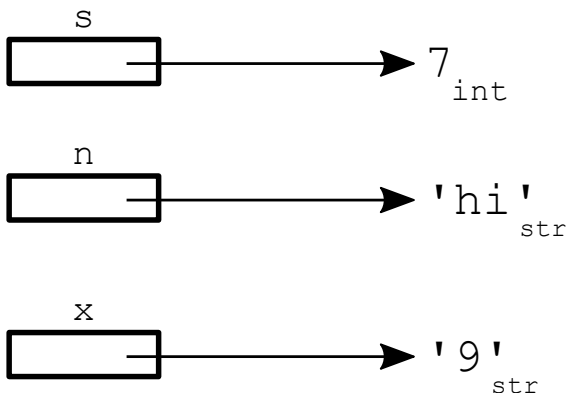
A: `123` is an `int` number, and `'123'` is a string length 3, made of 3 digit chars

`int` `str` Variables

Suppose we set up these three variables

```
>>> s = 7
>>> n = 'hi'
>>> x = '9'
```

Here is what memory looks like. In Python, each value in memory is tagged with its type - here `int` or `str`.



Python Does Not Deduce Type from Variable Name

Normally we follow the convention that a variable named `s` points to a string. This is a good convention, allowing people reading the code get the right impression of what the variable stores. We always follow this convention in our example code, so students naturally get the impression that it's some sort of rule. As if Python knows the value is a string because the variable name is `s`.

In fact, Python does not have a rule that a certain variable name must point to a certain type. To Python, the variable name is just a label of that variable used to identify it within the code. Python's attitude to the variable name is like: this is the name my human uses for this variable.

The type comes from the value at the end of the arrow, such as `7` (int) or `'Hello'` (str).

Therefore, we were contrary for this example, picking names that go against the conventions to show that Python does not read anything special into the variable name.

Type = Destiny

Python uses the type of a value to guide operations on that value. We can see this by watching the behavior of the `+` operators. Think about what the result will be for the expressions like `s + s`.

For each variable, Python follows the arrow to get the value to use, and each value is tagged with its type. What is the result for the expressions like `s + s` below?

```
>>> s = 7
>>> n = 'hi'
>>> x = '9'
>>>
>>> s + s
???
```

```
>>> n + n
???
```

```
>>> x + x
???
```

What are the ??? above?

Show

Type Conversions - `int()` `str()`

- Integer type is `int`
- String type is `str`
- Each type name is also the name of a conversion function:
- `int(xxx)` takes in string (or other) value, converts to `int`
e.g. `int('123')` -> `123`
- `str(xxx)` takes in `int` (or other) value, converts to `str`
e.g. `str(77)` -> `'77'`
- Works for other types we will see later too: `float()`, `list()`, `bool()`

```
>>> # e.g. text out of a file - a string
>>> # convert str form to int
>>> text = '123'
>>> int(text)
123
>>>
>>> # works the other way too
>>> str(123)
'123'
>>>
>>> # append int to str - error like this
>>> 'score:' + 13
TypeError: can only concatenate str (not "int")
to str
>>>
>>> # use str() to append int to str
>>> 'score:' + str(13)
'score:13'
>>>
```

Example/Exercise sum_digits()

'12abc3' -> 6

This example combines the accumulate pattern and str/int conversion.

sum_digits(s): Given a string s. Consider the digit chars in s. Return the arithmetic sum of all those digits, so for example, '12abc3' returns 6. Return 0 if s does not contain any digits.

> [sum_digits\(\)](#).

sum_digits() Starter

Here's the rote parts of sum_digits() you can start with:

```
def sum_digits(s):
    sum = 0

    for i in range(len(s)):
        # use s[i]
        pass

    return sum
```

sum_digits() Solution

```
def sum_digits(s):
    sum = 0
    for i in range(len(s)):
        if s[i].isdigit():
            # str '7' -> int 7
            num = int(s[i])
            sum += num
    return sum
```

int div Indexing - Skip for Today

Today - skipping down to "modulo" below, do int-div later.

1. Division / always produces float

```
>>> 7 / 2
3.5
>>> 8 / 2
4.0
```

2. Cannot use float for indexing or range()

```
>>> range(7 / 2)
TypeError: 'float' object cannot be interpreted
as an integer
```

3. int-division operator // rounds down to produce int. We'll look at later.

```
>>> 7 // 2
3
>>> 8 // 2
4
```

Example: right_left()

> [right_left\(\)](#).

'aabb' -> 'bbbbaaaa'

`right_left(s)`: We'll say the midpoint of a string is the len divided by 2, dividing the string into a left half before the midpoint and a right half starting at the midpoint. Given string `s`, return a new string made of 2 copies of right followed by 2 copies of left. So 'aabb' returns 'bbbbaaaa'.

Where do you cut the string `Python`?

The back half begins at index 3. The length is 6, so an obvious approach is to divide the length by 2. This actually does not work, and leads to a whole story.

(len 6)

P	y	t	h	o	n
0	1	2	3	4	5

Dividing length by 2 leads to an error...

```
>>> s = 'Python'
>>> mid = len(s) / 3
>>> mid
3.0
>>> s[mid:]
TypeError: slice indices must be integers ...
>>>
```

Recall: int vs. float

There are two number systems in the computer **int** for whole-number integers, and **float** for floating point numbers. The math operators `+` `-` `*` `**` work for both number types, so for many day-to-day computations the int/float distinction is not important. However, in this case, we are running into the issue that float does not work for indexing:

1. Use int for indexing, float does not work (e.g. 3.0 above)
2. The division operator `/` produces a float, even if the math comes out even

```
>>> 7 / 2
3.5
>>> 6 / 2
3.0
```

Int Division `//` Produces int

Python has a separate "int division" operator. It does division and discards any remainder, rounding the result down to the next integer.

```
>>> 7 // 2
3
>>> 6 // 2
3
>>> 9 // 2
4
>>> 8 // 2
4
>>> 94 // 10
9
>>> 102 // 4
25
```

This will work for `right_left()` - computing the int index where the right half begins, rounding down if the length is odd.

Aside: Left Half of Image

Suppose the width of an image is in a variable `width`, with a value like 100 or 125. What is the for-loop to generate the x values for the left half of the image?

```
# NO .. int/float issue
for x in range(width / 2):
    # use x in here
```

```
# YES
for x in range(width // 2):
    # use x in here
```

```
# e.g. width = 100: range is 0..49, since width
// 2 -> 50
```

Solve `right_left()`

```
'aabb' -> 'bbbbaaaa'
```

> [right_left\(\)](#).

Now solve `right_left()`. Use `//` to compute int "mid", rounding down in the case of a string of odd width. Our solution uses a decomp-by-var strategy, storing intermediate values in variables.

`right_left()` Solution

With the decomp-by-var strategy: solve a sub-part of the problem, storing the partial result in a variable with a reasonable name. Use the var on later lines. This is decomposition at a small scale - breaking a long line into pieces. Also the variable names make it nicely readable.

```
def right_left(s):
    midpoint = len(s) // 2
    left = s[:midpoint]
    right = s[midpoint:]
    return right + right + left + left
```

`right_left()` Without Decomp By Var - Yikes!

Here is the solution without the variables - yikes!

```
def right_left(s):
    return s[len(s) // 2:] + s[len(s) // 2:] + s[:len(s) // 2] +
s[:len(s) // 2]
```

The solution is shorter - just one line long, but the decomp-var version is more readable. Readability is not to help some other person, it's to help yourself. Bugs are when the code does something unexpected, and readability is at the core of that.

Modulo, Mod % Operator

The "mod" operator `%` is essentially the remainder after int division. So for example `(23 % 10)` yields 3 — divide 23 by 10 and 3 is the leftover remainder. The formal word for this is "modulo", but the word is often shortened to just "mod". The mod operator makes the most sense with positive numbers, so avoid negative numbers in modulo arithmetic.

- Mod by n: the remainder after dividing by n

- Mod by n: result is always in range 0..n-1
- Mod by 0 is an error, just like divide by 0
- If mod by n results in 0: the division came out evenly
Zero remainder
- Use non-negative integers with mod
It's possible to make negative work, but not useful

```
>>> 56 % 10
6
>>> 57 % 10
7
>>> 60 % 10      # 0 -> divides evenly
0
>>> 57 % 5
2
>>> 55 % 5
0
>>> 55 % 0
ZeroDivisionError: integer division or modulo by
zero
>>>
```

Mod - Even vs. Odd

A simple use of mod is checking if an int is even or odd - $n \% 2$ is 0 -> n is even, otherwise odd. It's common to use mod like this to, say, color every other row of a table green, white, green, white .. pattern. (See next example)

```
>>> 8 % 2
0
>>> 9 % 2
1
>>> 10 % 2
0
>>> 11 % 2
1
```

Example crazy_str()

crazy_str(s): Given a string s, return a crazy looking version where the first char is lowercase, the second is uppercase, the third is lowercase, and so on. So 'Hello' returns 'hElLo'. Use the mod % operator to detect even/odd index numbers.

'Hello' -> 'hElLo'

index:	0	1	2	3	4
	lower	upper	lower	upper	lower ...

even index: lower

odd index: upper

> [crazy_str\(\)](#).

crazy_str() Solution

```
def crazy_str(s):
    result = ''
    for i in range(len(s)):
        if i % 2 == 0: # even
            result += s[i].lower()
        else:
            result += s[i].upper()
    return result
```

File Processing - crazycat example

We'll use the crazycat example to demonstrate files, file-processing, printing, standard output, and functions.

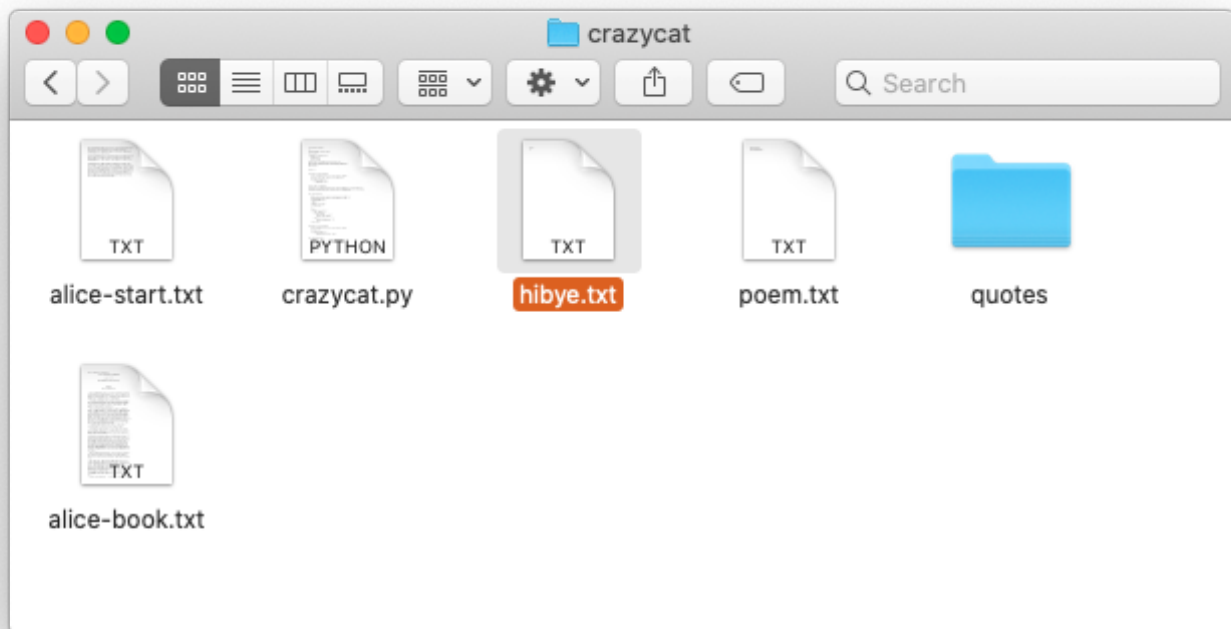
[crazycat.zip](#)

What is a Text File?

- "text file", aka "plain text file"
- **Extremely common** way to store/exchange data on computers
- Very old (teletype) .. and used up through today
- A text file is a series of **lines**
- Each line is a series of **chars** ending with a ' \n ' char
- Special char: ' \n ' is called the "newline" char
- ' \n ' is like hitting the "return" or "enter" key on your keyboard
- Aside: a few other chars can appear instead of ' \n ', detailed below

hibye.txt Text File Example

The file named "hibye.txt" is in the crazycat folder. What is a file? A file on the computer has a name and stores a series of bytes. The file data does not depend on the computer being switched on. The file is said to be "non-volatile".



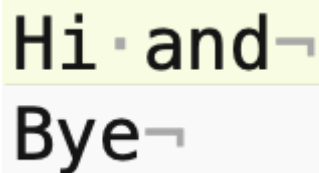
bibye.txt Contents

Text file: series of lines, each line a series of chars, each line marked by ' \n ' at end

The hbye.txt file has 2 lines, each line has a ' \n ' at the end. The first line has a space, aka ' ', between the two words. Here is the complete contents:

```
Hi and  
bye
```

Here is what that file looks like in an editor that shows little gray marks for the space and \n (like show-invisibles mode in word processor):



```
Hi · and↵  
Bye↵
```

In Fact the contents of that file can be expressed as a Python string - see how the newline chars end each line:

```
'Hi and\nbye\n'
```

Backslash Chars in a String

Use backslash \ to include special chars within a string literal. Note: different from the regular slash / on the ? key.

```
\n  newline char  
\'  single quote  
\\" double quote  
\  backslash char
```

```
s = 'isn\t'  
# or use double quotes  
s = "isn't"
```

How many chars? How many bytes?

How many chars are in that file (each `\n` is one char)?

There are 11 chars. The latin alphabet A-Z chars like this take up 1 byte per char. Characters in other languages take 2 or 4 bytes per char. Use your operating system to get the information about the `hibye.txt` file. What size in bytes does your operating system report for this file?

So when you send a 50 char text message .. that's about 50 bytes sent on the network + some overhead. Text data like this uses very few bytes compared to sound or images or video.

Aside: Detail About Line Endings

In the old days, there were two chars to end a line. The `\r` "carriage return", would move the typing head back to the left edge. The `\n` "new line" would advance to the next line. So in old systems, e.g. DOS, the end of a line is marked by two chars next to each other `\r\n`. On Windows, you will see text files with this convention to this day. Python code largely insulates your code from this detail - the `for line in f` form shown below will go through the lines, regardless of what line-ending they are encoded with.

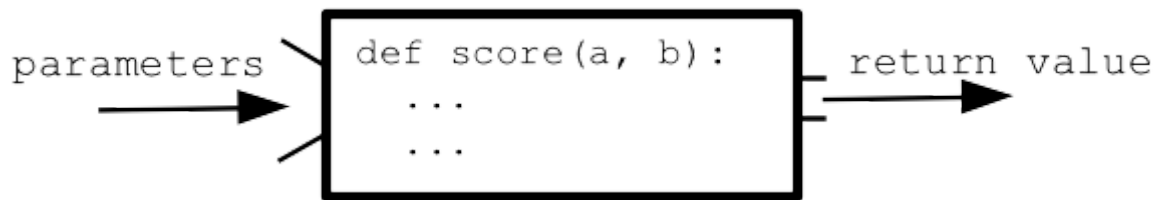
Recall: Function Data = Parameters and Return

Q: How does data flow between the functions in your program?

A: **Parameters** and **Return value**

Parameters carry data from the caller code into a function when it is called. The return value of a function carries data back to the caller.

This is the key data flow in your program. It is 100% the basis of the Doctests. It is also the basis of the old black-box picture of a function

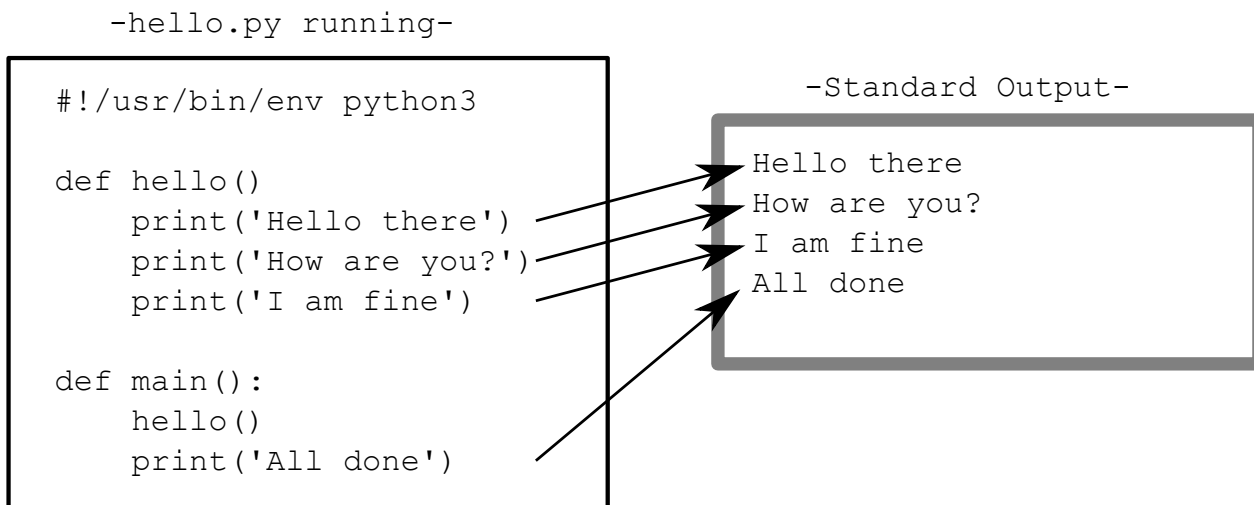


"Standard Output" Text Area

BUT .. there is an additional, parallel output area for a program, shared by all its functions.

There is a text area known as **Standard Output** associated with every run of a program. By default standard output is made of text, a series of text lines, just like a text file. Any function can append a line of text to standard out by calling the `print()` function, and conveniently that text will appear in the terminal window hosting that run of python code. the standard output area works in other computer languages too, and each language has its own form of the `print()` function.

Here we see the `print()` output from calling the `main()` function in this example:



`print()` function

See guide: [print\(\)](#).

- Python `print()` function
- Prints text lines to the standard output area
- In the `>>>` interpreter, `print()` output appears in the interpreter

- Takes a number of items, separated by commas
- Converts each item to string form
- Places a ' \n ' at the end of the line
- Note that strings do not have quotes around them in output
- print() can be used for debugging a little, see state of variables
- Try print() in the interpreter, see its output right there

```
>>> print('hello there')
hello there
>>> print('hello', 123, '!')
hello 123 !
>>> print(1, 2, 3)
1 2 3
```

Data out of function: return vs. print

Return and print() are both ways to get data out of a function, so they can be confused with each other. We will be careful when specifying a function to say that it should "return" a value (very common), or it should "print" something to standard output (rare). Return is the most common way to communicate data out of a function, but below are some print examples.

Crazycat Program example

This example program is complete, showing some functions, Doctests, and file-reading.

[crazycat.zip](#)

1. Try "ls" and "cat" in terminal

See guide: [Command line](#)

See guide: [File Read/Write](#)

Open the crazycat project in PyCharm. Open a terminal in the crazycat directory (see the [Command Line](#) guide for more information running in the terminal).

Terminal commands - work in both Mac and Windows. When you type command in the terminal, you are typing command directly to the operating system that runs your computer - Mac OS, or Windows, or Linux.

`pwd` - print out what directory we are in

`ls` - see list of filenames ("dir" on older Windows)

`cat filename` - see file contents ("type" on older Windows)

```
$ ls
alice-book.txt  hibye.txt          quotes
crazycat.py    poem.txt
$ cat poem.txt
Roses Are Red
Violets Are Blue
This Does Not Rhyme
$
```

2. Run crazycat.py with filename

- The crazycat.py program does "cat" but implemented in Python
Demonstrating how to read lines of a text file and print them out
- Use the tab-key to autocomplete filenames
- The standard out of a program is typically printed to the terminal

```
$ python3 crazycat.py poem.txt
Roses Are Red
Violets Are Blue
This Does Not Rhyme
$ python3 crazycat.py hibye.txt
Hi and
bye
$
```

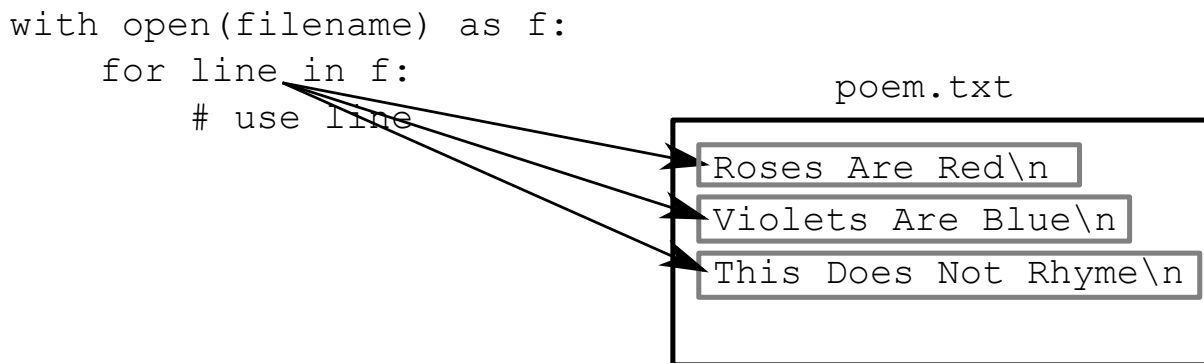
Standard File-Read Code v1

Say the variable `filename` holds the name of a file as a string, like `'poem.txt'`. Here is the standard code to read through the lines of the file:

```
with open(filename) as f:
    for line in f:
        # use line
    ...
```

File-Read Picture

Here is how the variable "line" behaves for each iteration of the loop:



- `for line in f`
- On the first iteration, `line` is set to point to a string of the first line of the file
- On the second iteration, the second line, and so on
- Each line string has a `'\n'` newline char at its end
- Memory efficient
Only holds one line of text in memory at a time
Even if the file is extremely large
- other forms of `open()`:
 - `open(filename)` - open for reading, most common, CS106A uses this
 - `open(filename, 'r')` - same as above, 'r' denotes reading
 - `open(filename, 'w')` - open for writing
 - `open(filename, encoding='utf-8')` - specify unicode encoding (later)

Background: `s.strip()` Function

The string `s.strip()` function, removes whitespace chars like space and newline from the beginning and end of a string and returns the cleaned up string. Here we use it as an easy way to get rid of the newline.

```
>>> s = '  hello there\n'
>>> s
'  hello there\n'
>>> s.strip()
'hello there'
```

Addition: `line = line.strip()`

Each line string inside the `for line in f` loop has the `'\n'` newline char at its end. Half the time, this newline char makes no difference to anything, and half the time it ends up getting in the way.

Therefore, we will make a habit of adding `line = line.strip()` in the loop which removes the `'\n'` char so we don't have to think about it.

Standard File Read Code v2 - `line.strip()`

Here is the file read code with the `line.strip()` added. For CS106A, we will always write it this way, so we never see the `'\n'`.

```
with open(filename) as f:
    for line in f:
        line = line.strip()    # remove \n
        # use line
```

If some CS106A problem asks you to read all the lines of a file, you could paste in the above.

3. Look at `print_file_plain()` Code

Back to crazycat example - look at the code.

This command line we saw earlier calls the `print_file_plain()` function below, passing in the string `'poem.txt'` as the filename.

```
$ python3 crazycat.py poem.txt
Roses Are Red
Violets Are Blue
This Does Not Rhyme
```

Here is the `print_file_plain()` function that implements the "cat" feature - printing out the contents of a file. This code reads every line out of a file, printing each line to standard output.

```
def print_file_plain(filename):
    """
    Given a filename, read all its lines and
    print them out.
    This shows our standard file-reading loop.
    """
    with open(filename) as f:
        for line in f:
            line = line.strip()    # remove \n
            print(line)
```

4. Run With -crazy Command Line Option

The `main()` function looks for `'-crazy'` option on the command line. We'll learn how to code that up soon. For now, just know that `main()` calls the `print_file_crazy()` function which calls the `crazy_line()` helper.

Here is command line to run with `-crazy` option

```
$ python3 crazycat.py -crazy poem.txt
rOsEs aRe rEd
vIoLeTs aRe bLuE
tHiS DoEs nOt rHyMe
```

How does the program produce that output?

5. Recall crazy_str(s) Function

- Recall the crazy_str() helper function
- Wrote this code earlier
- Returns version of input string with lower/upper char pattern
- Standard black-box black-box design
 - Takes in string parameter
 - Returns result string
 - Does not use print()
 - Has Doctests

```
def crazy_str(s):  
    """  
    Given a string s, return a crazy looking  
    version where the first  
    char is lowercase, the second is uppercase,  
    the third is  
    lowercase, and so on. So 'Hello' returns  
    'hElLo'.  
    >>> crazy_str('Hello')  
    'hElLo'  
    >>> crazy_str('@xYz!')  
    '@XyZ!'  
    >>> crazy_str('')  
    ''  
    """  
    result = ''  
    for i in range(len(s)):  
        if i % 2 == 0:  
            result += s[i].lower()  
        else:  
            result += s[i].upper()  
    return result
```

6 - Print-Crazy Plan

1. Read each line from file
2. Pass each line into `crazy_str()` function to compute crazy form
3. Print the crazy form

7. `print_file_crazy()` Code

The code is similar to `print_file_plain()` but passes each line through the `crazy_str()` function before printing. Think about the flow of data in the code below.

```
def print_file_crazy(filename):  
    """  
        Given a filename, read all its lines and  
        print them out  
        in crazy form.  
    """  
    with open(filename) as f:  
        for line in f:  
            line = line.strip()  
            crazy = crazy_str(line)  
            print(crazy)  
            # could combine above - dense:  
            # print(crazy_str(line))
```

Experiments

1. Run on `alice-book.txt` - 3600 lines. The file for-loop rips through the data in a fraction of a second. You can get a feel for how your research project could use Python to tear through some giant text file of data.

```
python3 crazycat.py -crazy alice-book.txt
```

2. Shorten the `print()` in the loop to one line, as below. Describe the sequence of things that happens to each line:

```
print(crazy_str(line))
```


3. (very optional) Try removing the `line = line.strip()`. What happens to the output? What is happening: the line has a `'\n'` at its end. The `print()` function also adds a newline at the end of what it prints.

Optional > Trick

Try running this way, works on all operating systems:

```
$ python3 crazycat.py -crazy alice-book.txt > capture.txt
```

What does this do? Instead of printing to the terminal, it captures standard output to a file "capture.txt". Use "ls" and "cat" to look at the new file. This is a super handy way to use your programs. You run the program, experimenting and seeing the output directly. When you have a form you, like use `>` once to capture the output. Like the pros do it!