WHERE WERE WE?

- Backwards (in 3 ways)
- Strings
WHERE WERE WE?

- Backwards (in 3 ways)
- Strings
Backwards (1)

- Reverse the samples:
  
  ```python
  recipe 60
  def backwards( fileName ) :
    source = makeSound( fileName )
    target = makeSound( fileName )
    sourceIndex = getLength( source )
    targetLength = sourceIndex + 1
    for targetIndex in range( 1, targetLength ):
      sourceValue = getSampleValueAt( source, sourceIndex )
      setValueAt( target, targetIndex, sourceValue )
      sourceIndex = sourceIndex - 1
    return target
  ```
Backwards (2): mirror

- Reverse the samples:
  - recipe 61
  - same algorithm as recipe 19 (pictures)

```python
def mirror(sound):
mirrorPoint = getLength(sound) / 2

for sampleOffset in range(1, mirrorPoint - 1):
    sampleLater = getSampleObjectAt(sound, mirrorPoint+sampleOffset)
    sampleBefore = getSampleObjectAt(sound, mirrorPoint-sampleOffset)
    value = getSample(sampleBefore)
    setSample(sampleLater, value)

return sound
```
Backwards (3): mirror using separate functions

- What are some pieces we can learn from:
  - Algorithms from 60 & 61

- What parts do we want to re-use to make this recipe of recipes from?
  - 60 takes a filename, returns a sound
  - 61 takes a sound, (our version) returns a sound

- What are some ways we can think of using backwards sounds?
  - create backwards sounds from anywhere in a sound
  - put backwards sound anywhere in new sound
Backwards (3):

backwardsSection( .... )

```
def backwardsSection( source, takeFrom, length, target, putTo ) :
    # this is our general purpose function
    sourceIndex = min(takeFrom + length, getLength( source))
    targetLength = min(putTo + length, getLength(target) )

    # loop through forward through the target and backwards through the source
    for targetIndex in range( putTo, targetLength ):
        sourceValue = getSampleValueAt( source, sourceIndex )
        setSampleValueAt( target, targetIndex, sourceValue )
        sourceIndex = sourceIndex - 1
        if sourceIndex < 1 :
            return target

    return target
```

```
def min( param1, param2 ) :
    # returns the lessor of two parameters
    if param1 < param2 :
        return param1
    else :
        return param2
```
Backwards (3):
recipe 60 & 61 revisited

def backwards( filename ) :
    # the equivalent of recipe 60
    source = makeSound( filename )
    target = makeSound( filename )
    sourceLength = getLength( source )
    return backwardsSection( source, 1, sourceLength, target, 1)

def mirror( sound ) :
    # the equivalent of recipe 61
    sourceLength = getLength( sound )
    return backwardsSection( sound, 1, sourceLength/2, sound, (sourceLength/2) + 1 )

def mirrorFile( filename ) :
    # same as mirror, but uses filename
    source = makeSound( filename )
    target = makeSound( filename )
    sourceLength = getLength( source )
    return backwardsSection( source, 1, sourceLength/2, target, (sourceLength/2) + 1 )

def revMirror( sound ) :
    # like mirror except the first half is reversed and second half is forward
    # equivalent to mirror(backwards() )
    sourceLength = getLength( sound )
    return backwardsSection( sound, (sourceLength/2) + 1, sourceLength/2, sound, 1 )
Backwards (3):
chop sound, reverse alternates

def revFragments( source, numOfFragments ) :  
    # chops sound into numOfFragments, reverse every other one 
    target = makeEmptySound( getLength(source), int(getSamplingRate( source) ) ) 
    sourceLength = getLength( source ) 
    fragLength = sourceLength / numOfFragments 
    start = 1 
    
    # step for every other fragment 
    for count in range(1, numOfFragments + 1, 2) : 
        target = backwardsSection( source, start, fragLength, target, start ) 
        start = start + (fragLength * 2) 

    return target
WHERE WERE WE?

- Backwards (in 3 ways)
- Strings
Strings

using strings to write HTML
New programming syntax and concepts

- Up until now, we’ve had a small set of programming elements we’ve worked with:
  - Assignment, print, for (with and without range()), if
- We’re halfway through the class, so we’re going to start pulling back the curtains a little and show what’s behind the scenes
Text

- Text is the universal medium
  - We can convert any other media to a text representation.
  - We can convert between media formats using text.
  - Text is simple.
- Text is usually processed in an array—a long line of characters
- We refer to one of these long line of characters as a string.
Strings

- Strings are defined with quote marks.
- Python actually supports three kinds of quotes:
  >>> print 'this is a string'
  this is a string
  >>> print "this is a string"
  this is a string
  >>> print """"this is a string"""
  this is a string
- Use the right one that allows you to embed quote marks if you want
  >>> phrase = "Monica's cat."
  >>> print phrase
  Monica’s cat.
Why would you want to use triple quotes?

- To have long quotations with returns and such inside them.

```python
def aLongString():
    return """This is a long string""
```

```python
>>> print aLongString()
This is a long string
```
Encodings for strings

- Strings are just arrays of characters
- In most cases, characters are just single bytes.
  - The ASCII encoding standard maps between single byte values and the corresponding characters
- More recently, characters are two bytes.
  - Unicode uses two bytes per characters so that there are encodings for glyphs (characters) of other languages
  - Java uses Unicode. The version of Python we are using is based in Java, so our strings are actually using Unicode.
There are more characters than we can type

- Our keyboards don’t have all the characters available to us, and it’s hard to type others into strings.
  - Backspace?
  - Return?
  - ? ±
- We use backslash escapes to get other special characters
Backslash escapes

- “\b” is backspace
- “\n” is a newline (like pressing the Enter key)
- “\t” is a tab
- “\uXXXX” is a Unicode character, where XXXX is a code and each X can be 0-9 or A-F.

- [http://www.unicode.org/charts/](http://www.unicode.org/charts/)
- Must precede the string with “u” for Unicode to work
Testing strings

>>> print "hello\ttthere\nMark"
hello there
Mark

>>> print u"\uFEED"

>>> print u"\u03F0"

>>> print "Thix\bs is\na\btest"
Manipulating strings

We can add strings and get their lengths using the kinds of programming features we’ve seen previously.

```python
>>> helloStr = "Hello"
>>> print len(helloStr)
5
>>> markStr = ", Mark"  <---- has a space after the comma
>>> print len(markStr)
6
>>> print helloStr + markStr
Hello, Mark
>>> print len(helloStr + markStr)
11
```
Getting parts of strings

- We use the square bracket “[]” notation to get parts of strings.
- `stringVariable[n]` gives you the $n^{th}$ character in the string (but keep in mind the first one is the zero-ith).
- `string[n:m]` gives you the characters indexed by $n$ through (but not including) index $m$. 
Getting parts of strings

```python
>>> helloStr = "Hello"
>>> print helloStr[1]
e
>>> print helloStr[0]
H
>>> print helloStr[2:4]
ll
```
Start and end indices are assumed if not there

```python
>>> print helloStr
Hello
>>> print helloStr[:4]
Hell
>>> print helloStr[3:]
lo
>>> print helloStr[:]
Hello
```
WHERE WERE WE?

- Backwards (in 3 ways)
- Strings
Debug some code

Due next Wednesday @ 10:00 AM
Coming Attractions

- Wednesday:
  - HW 6 due 10:00 AM
- Friday:
  - Group Project due 2:00 PM
- Next Monday:
  - read Chapter 10 & 11 through section 11.3
  - Quiz due 10:00 AM