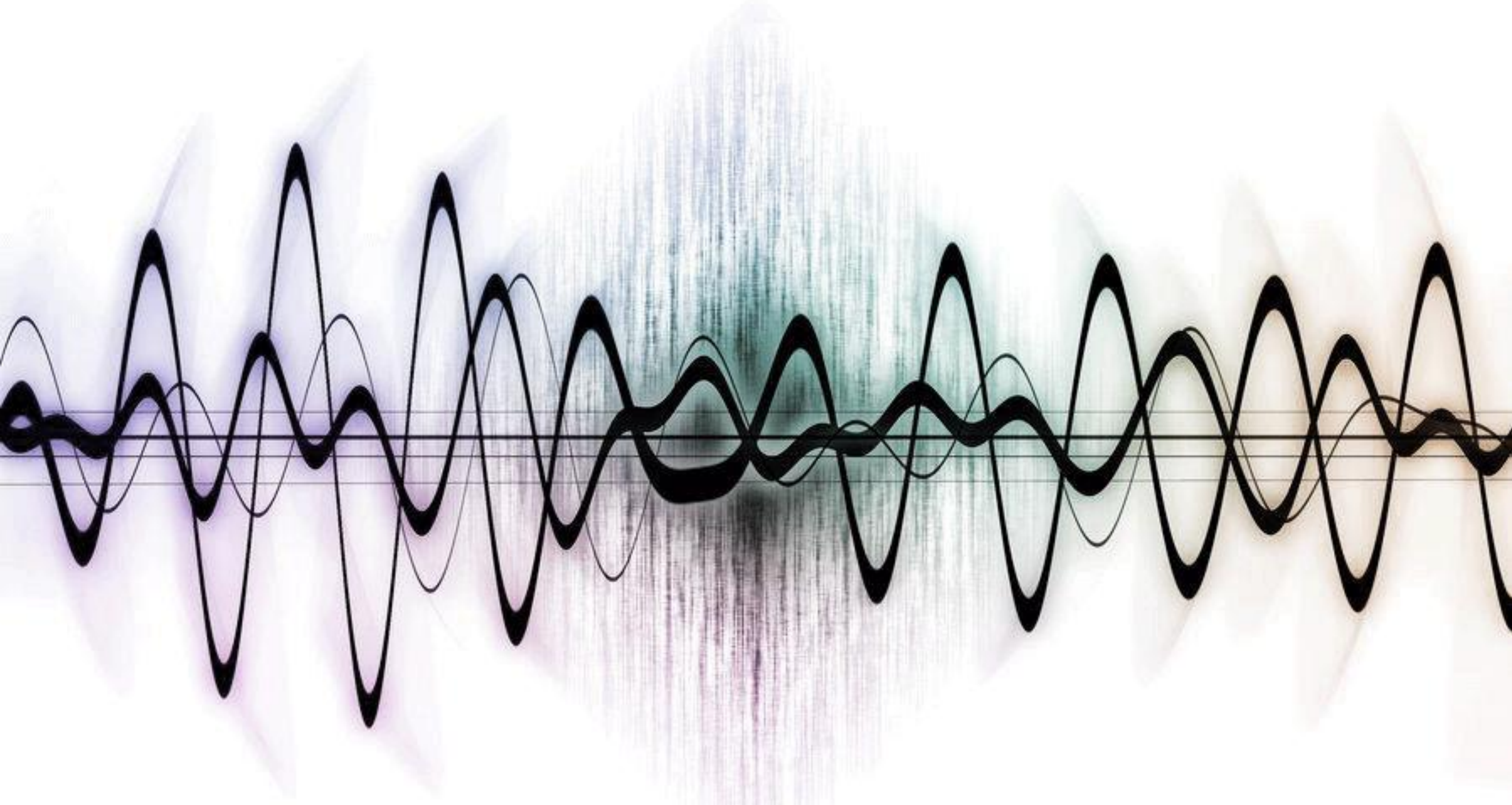
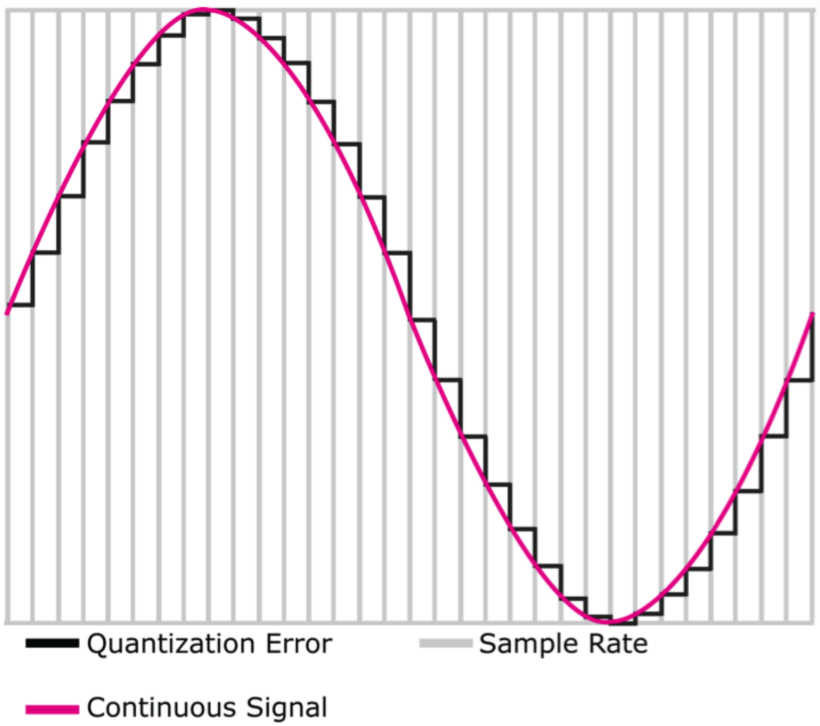


Tinkering Audio III: Construction of Melodies

Creative Computing: Tinkering – Lecture 10 – Michael Scott

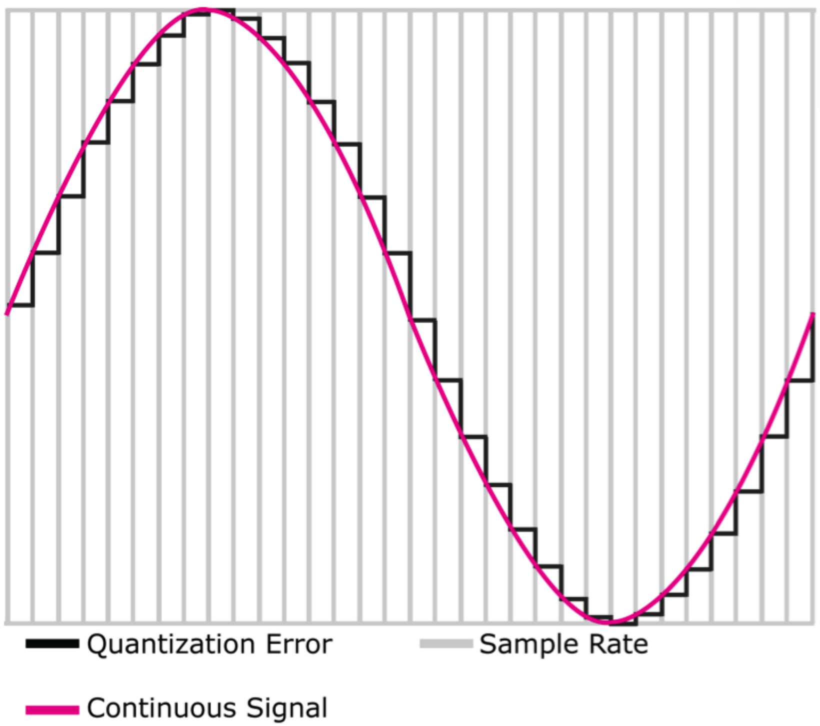
Recap on Last Week

- Bit-depth is important for representing displacement
  - Amplitude is displacement at a certain point (usually the crest and the trough)
  - Available range of volume often depends on the available hardware
  - Distance between two levels, however, depends on how bits are allocated



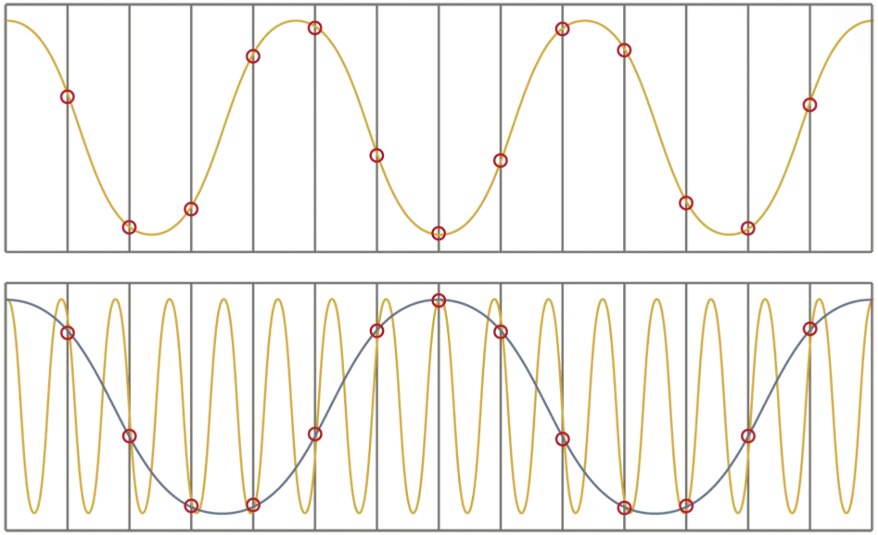
Recap on Last Week

- Bit-depth is important for representing displacement
  - Error arises when a displacement cannot be represented
  - Clipping occurs when maximum volume is exceeded, but this can be avoided with a sufficient bit-depth and normalization procedure



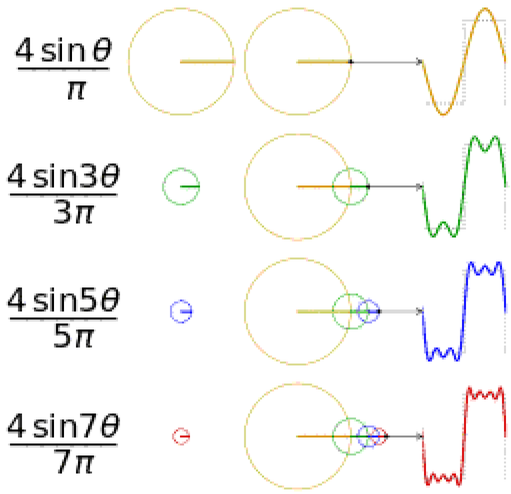
Recap on Last Week

- Sample rate is important for representing frequency
  - Aliasing occurs when sample rate is too low to represent the intended frequency
  - The wrong frequency is produced
  - Use Nyquist Theorem to determine the sample rate needed for a particular frequency



Recap on Last Week

- Signals can be combined through simple summation to create more complex signals
  - These sin() signals approximate a square wave

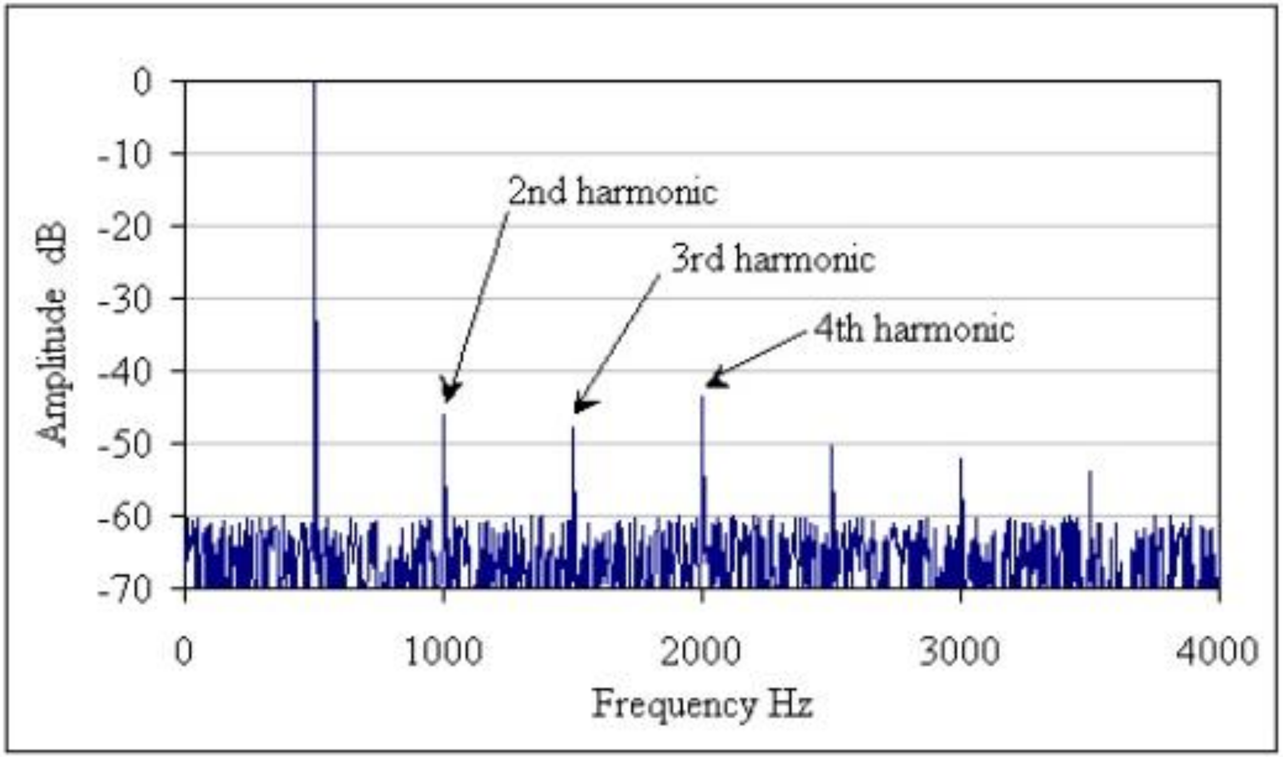


Recap on Last Week

- We can mix sounds
  - We even know how to change the volumes of the two sounds, even over time (e.g., fading in or fading out)

- We can add sine (or other) waves together to create kinds of instruments / sounds that do not physically exist, but which sound interesting and complex

Recap on Last Week



The Harmonics of 500 Hz

Recap on Last Week

- Splicing gets its name from literally cutting and pasting pieces of magnetic tape together
  - Use generalisable clip and copy functions to construct sounds from other sounds

```
def clip(source, start, end):
  target = makeEmptySound(end - start)
  tIndex = 0
  for sIndex in range(start, end):
    value = getSampleValueAt(source, sIndex)
    setSampleValueAt(target, tIndex, value)
    tIndex = tIndex + 1
  return target
```

```
def copy(source, target, start):
  tIndex = start
  for sIndex in range(0, getLength(source)):
    value = getSampleValueAt(source, sIndex)
    setSampleValueAt(target, tIndex, value)
    tIndex = tIndex + 1
```

Learning Objectives

By the end of this session, you will be able to:

- **Explain how** to implement echoes, envelopes, and a basic additive synthesiser
- **Recognise** the parallels between sampling and scaling sound with sampling and scaling images
- **Write** a basic function to manipulate and combine sine waves according to their frequency, amplitude, attack-time, sustain-time, and decay-time
- **Explain** parsing **and how** to integrate notation into a synthesiser
- **Integrate** randomness into Python code using the random library

Tinkering Audio

ECHOES

A Function for Adding Two Sounds

```
def addSoundInto(sound1, sound2):
  for sampleNmr in range(0, getLength(sound1)):
    sample1 = getSampleValueAt(sound1, sampleNmr)
    sample2 = getSampleValueAt(sound2, sampleNmr)
    setSampleValueAt(sound2, sampleNmr, sample1 + sample2)
```

Notice that this adds sound1 and sound2 by adding sound1 ***into*** sound2

Adding Sounds with a Delay

```
def makeChord(sound1, sound2, sound3):
  for index in range(0, getLength(sound1)):
    s1Sample = getSampleValueAt(sound1, index)
    setSampleValueAt(sound1, index, s1Sample )
    if index > 1000:
      s2Sample = getSampleValueAt(sound2, index - 1000)
      setSampleValueAt(sound1, index, s1Sample + s2Sample)
    if index > 2000:
      s3Sample = getSampleValueAt(sound3, index - 2000)
      setSampleValueAt(sound1, index, s1Sample + s2Sample + s3Sample)
```

-Add in sound2 after 1000 samples

-Add in sound3 after 2000 samples

Note that in this version we're adding directly into sound1!

Creating an Echo

```
def echo(sndFile, delay):
  s1 = makeSound(sndFile)
  s2 = makeSound(sndFile)
  for index in range(delay, getLength(s1)):
    echo = 0.6*getSampleValueAt(s2, index-delay)
    combo = getSampleValueAt(s1, index) + echo
    setSampleValueAt(s1, index, combo)
  play(s1)
  return s1
```

This creates a delayed echo sound, multiplies it by 0.6 to make it fainter and then adds it into the original sound.

How the Echo Works

Top row is the samples of our sound. We're adding it to itself, but delayed a few samples, and multiplied to make it softer.

100	200	1000	-150	-350	200	500	10	-500	-1000	-350	25	-10	1000	...

Delay

100	200	1000	-150	-350	200	500	10	-500	-1000	-350	25	-10	1

* 0.6

60	120	600	-90	-210	120	300	6	-300	-600	-210	15	-6	6

+

100	200	1000	-90	-230	800	410	-200	-380	290	-344	-275	-610	890	...

Exercise

- Implement a double-echo algorithm, adapting the code in the previous slide

- echo(sndFile, initialDelay, nextDelay)

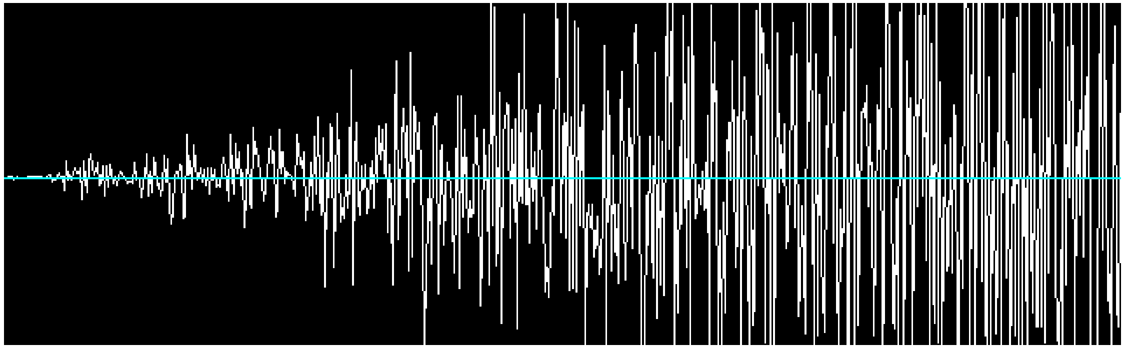
Echo with Feedback

```
def echo(sndFile, delay):
  s1 = makeSound(sndFile)
  s2 = makeSound(sndFile)
  for index in range(delay, getLength(s)):
```

- This function loads the sample in **twice**, then mixes the second instance into the first
- What happens if we load it **once** and mix it into itself?

Echo with Feedback

```
def echo(sndFile, delay):
  s = makeSound(sndFile)
  for index in range(delay, getLength(s)):
    echo = 0.6*getSampleValueAt(s, index-delay)
    combo = getSampleValueAt(s, index) + echo
    setSampleValueAt(s, index, combo)
  play(s)
  return s
```



Tinkering Audio

RESAMPLING

How sampling keyboards work

- They have a huge memory with recordings of lots of different instruments played at different notes
- When you press a key on the keyboard, the recording closest to the note you just pressed is selected, and then the recording is shifted to exactly the note you requested.
- The shifting is a generalization of algorithms of doubling and halving frequency (as follows)

Doubling the frequency

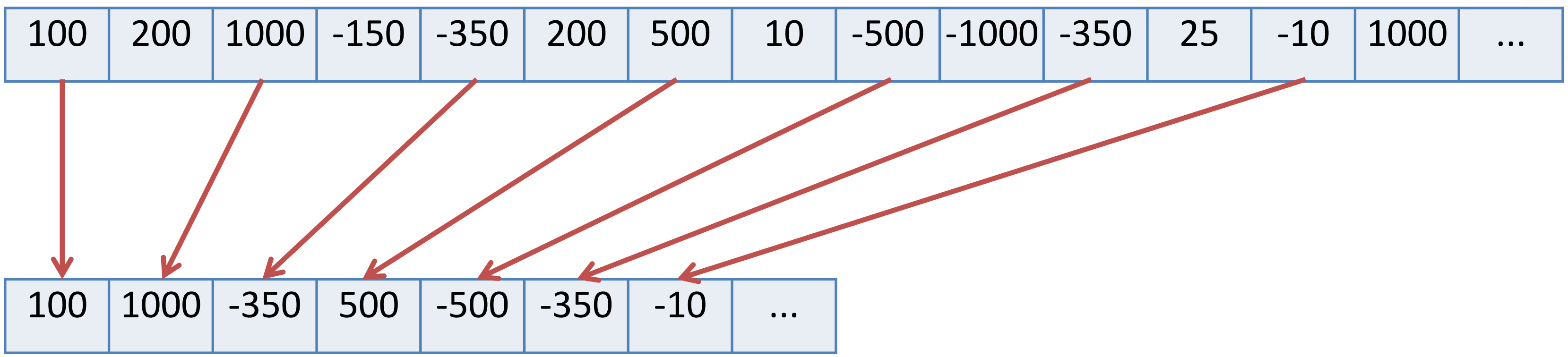
Why +1 here?

```
def double(source):
  len = getLength(source) / 2 + 1
  target = makeEmptySound(len)
  targetIndex = 0
  for sourceIndex in range(0, getLength( source), 2):
    value = getSampleValueAt( source, sourceIndex)
    setSampleValueAt( target, targetIndex, value)
    targetIndex = targetIndex + 1
  play(target)
  return target
```

Here's the piece that does the doubling

How doubling works

sourceIndex = 0, 2, 4, 6, 8, 10, ...



targetIndex = 0, 1, 2, 3, 4, 5, ...

Halving the frequency

```
def half(source):
  target = makeEmptySound(getLength(source) * 2)
  sourceIndex = 0
  for targetIndex in range(0, getLength( target)):
    value = getSampleValueAt( source, int(sourceIndex))
    setSampleValueAt( target, targetIndex, value)
    sourceIndex = sourceIndex + 0.5
  play(target)
  return target
```

Here's the piece that does the halving

How halving works

sourceIndex = 0, 0.5, 1, 1.5, 2, 2.5, 3, ...
int(sourceIndex) = 0, 0, 1, 1, 2, 2, 3, 3, ...

100	200	1000	-150	-350	200	500	10	-500	-1000	-350	25	-10	1000	...

100	100	200	200	1000	1000	-150	-150	-350	-350	200	200	500	500	...

targetIndex = 0, 1, 2, 3, 4, 5, ...

Can we generalize shifting a sound into other frequencies?

```
def shift(source, factor):
  target = makeEmptySound(getLength(source))
  sourceIndex = 0

  for targetIndex in range(0, getLength( target)):
    value = getSampleValueAt(source, int(sourceIndex))
    setSampleValueAt( target, targetIndex, value)
    sourceIndex = sourceIndex + factor

  play(target)
  return target
```

Why doesn't it work?

- It works for shifting down, but not for shifting up

```
>>> hello = makeSound("helloworld.wav")
>>> lowerHello = shift(hello, 0.6)
>>> higherHello = shift(hello, 1.5)
You are trying to access the sample at index: 44034, but the last valid index
is at 44032
The error value is:
Inappropriate argument value (of correct type).
An error occurred attempting to pass an argument to a function.
 in file C:\Program Files
(x86)\JES\jes\python\jes\core\interpreter\__init__.py, on line 157, in
function run
 in file C:\Program Files
(x86)\JES\jes\python\jes\core\interpreter\__init__.py, on line 202, in
function execute
 in file <input>, on line 1, in function <module>
 in file C:\Users\Ed\Desktop\jes\comp120_8_07_shift_broken.py, on line 6, in
function shift
 in file C:\Program Files (x86)\JES\jes\python\media.py, on line 346, in
function getSampleValueAt
ValueError:
Please check line 6 of C:\Users\Ed\Desktop\jes\comp120_8_07_shift_broken.py
>>>
```

Why doesn't it work?

```
def shift(source, factor):
  target = makeEmptySound(getLength(source))
  sourceIndex = 0

  for targetIndex in range(0, getLength( target)):
    value = getSampleValueAt(source, int(sourceIndex))
    setSampleValueAt( target, targetIndex, value)
    sourceIndex = sourceIndex + factor

  play(target)
  return target
```

- What goes wrong when factor > 1?
- Slack channel: **#comp120**

Three ways of fixing it: 1

```
def shift(source, factor):
    target = makeEmptySound(getLength(source))
    sourceIndex = 0

    for targetIndex in range(0, getLength( target)):
        value = getSampleValueAt( source, int(sourceIndex))
        setSampleValueAt( target, targetIndex, value)
        sourceIndex = sourceIndex + factor
        if sourceIndex > getLength(source):
            sourceIndex = 0

    play(target)
    return target
```

- What would shift(sound, 3) do?

Three ways of fixing it: 2

```
def shift(source, factor):
    target = makeEmptySound(getLength(source))
    sourceIndex = 0

    for targetIndex in range(0, getLength( target)):
        value = getSampleValueAt( source, int(sourceIndex))
        setSampleValueAt( target, targetIndex, value)
        sourceIndex = sourceIndex + factor
        if sourceIndex > getLength(source):
            break

    play(target)
    return target
```

- What would shift(sound, 3) do?

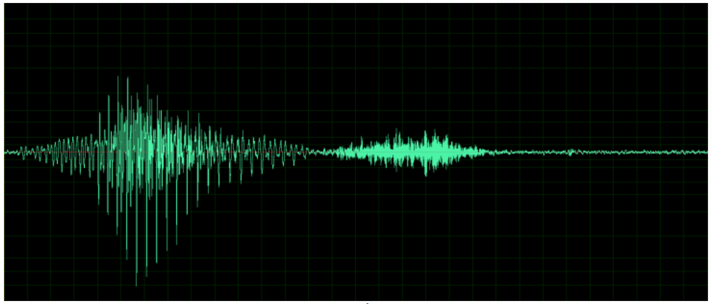
Three ways of fixing it: 3

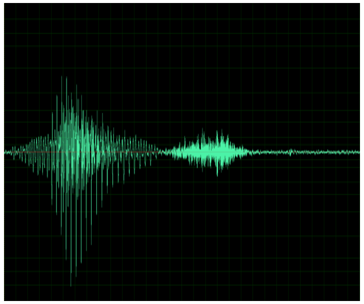
```
def shift(source, factor):
    target = makeEmptySound(int(getLength(source) / factor) + 1)
    sourceIndex = 0

    for targetIndex in range(0, getLength( target)):
        value = getSampleValueAt( source, int(sourceIndex))
        setSampleValueAt( target, targetIndex, value)
        sourceIndex = sourceIndex + factor

    play(target)
    return target
```

- What would shift(sound, 3) do?

Sampling as an Algorithm

- Think about the similarities between:
  - Halving the frequency of a sound
  - Scaling a picture up to twice the size
- Think about the similarities between:
  - Doubling the frequency of a sound
  - Scaling a picture down to half the size





Recall the Picture Copying Functions

```
def half(source):
  target = makeEmptySound(getLength(source) * 2)
  sourceIndex = 0
  for targetIndex in range(0, getLength( target)):
    value = getSampleValueAt( source, int(sourceIndex))
    setSampleValueAt( target, targetIndex, value)
    sourceIndex = sourceIndex + 0.5
  play(target)
  return target
```

```
def copyBarbsFaceLarger():
  # Set up the source and target pictures
  barbf=getMediaPath("barbara.jpg")
  barb = makePicture(barbf)
  canvasf = getMediaPath("7inX95in.jpg")
  canvas = makePicture(canvasf)
  # Now, do the actual copying
  sourceX = 45
  for targetX in range(100,100+((200-45)*2)):
    sourceY = 25
    for targetY in range(100,100+((200-25)*2)):
      color = getColor(
                getPixel(barb,int(sourceX),int(sourceY)))
      setColor(getPixel(canvas,targetX,targetY), color)
      sourceY = sourceY + 0.5
    sourceX = sourceX + 0.5
  show(barb)
  show(canvas)
  return canvas
```

Both of these functions implement a *sampling algorithm*

```
INITIALISE source index to 0
FOR EACH target index
    CONVERT source index to an integer
    GET the element at the integer source index
    PUT the element at the target index
    INCREMENT source index by 0.5
END FOR
RETURN target
```

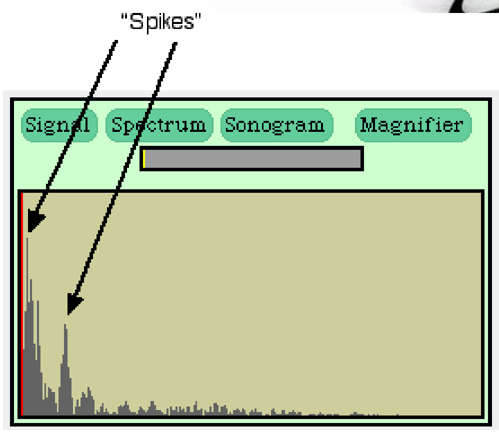
Tinkering Audio

SYNTHESISERS

Synthesizers

- We now have a very basic sampling synthesizer
- There are several other types of sound synthesis
  - **Additive**: works by adding together several sine waves or other waves
  - **Subtractive**: works by generating harmonically rich waves and then filtering them to shape their frequency content
  - **Frequency modulation (FM)**: works by rapidly altering the frequency of a generated tone
  - **Physical modelling**: works by simulating the vibrations in real-world objects such as strings, wind instruments, even human vocal cords
  - … Plus many others

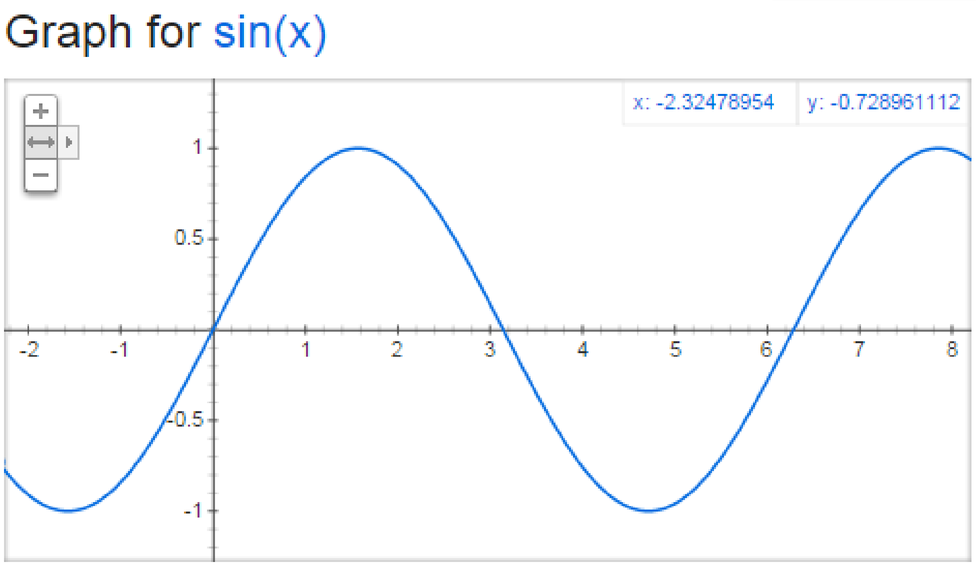
Adding sine waves to make something completely new



- We saw earlier that complex sounds (like the sound of your voice or a trumpet) can be seen as being a sum of sine waves
  – Any sound can be made by summing sine waves – Fourier's Theorem
- We can *create* complex sounds by summing sine waves
- These are sounds made by mathematics, by invention, not based on anything in nature

Basic idea: Build a sine wave

- "Hz" = "Cycles per second"
- If we want a 440 Hz sound wave, then we need one of these cycles every 1/440th of a second.



Our algorithm

First some calculations…

- frequency = 440Hz
- interval = length of 1 cycle in seconds = 1/frequency = 1/440 seconds
- samplesPerCycle = length of 1 cycle in samples
- = 1/440 seconds * 22050 samples/sec = 50.11 samples

Now for each output sample:

- Calculate sampleIndex / samplesPerCycle – this quantity increases by 1 on each cycle, i.e. every 1/440 seconds
- Multiply this by 2*pi to get an angle in radians
- Take the sine – this gives a value between -1 and +1
- Multiply by the desired amplitude and put it at sampleIndex

Our Code

```
def sineWave(freq, amplitude, length):
  # Create a blank sound
  buildSin = makeEmptySoundBySeconds(length)
  # Set constants
  samplingRate = getSamplingRate(buildSin)
  interval = 1.0 / freq
  samplesPerCycle = interval * samplingRate
  maxCycle = 2 * pi
  # Generate the sound
  for pos in range(getLength(buildSin)):
    rawSample = sin((pos / samplesPerCycle) * maxCycle)
    sampleVal = int(amplitude * rawSample)
    setSampleValueAt(buildSin, pos, sampleVal)
  return buildSin
```

Adding pure sine waves together

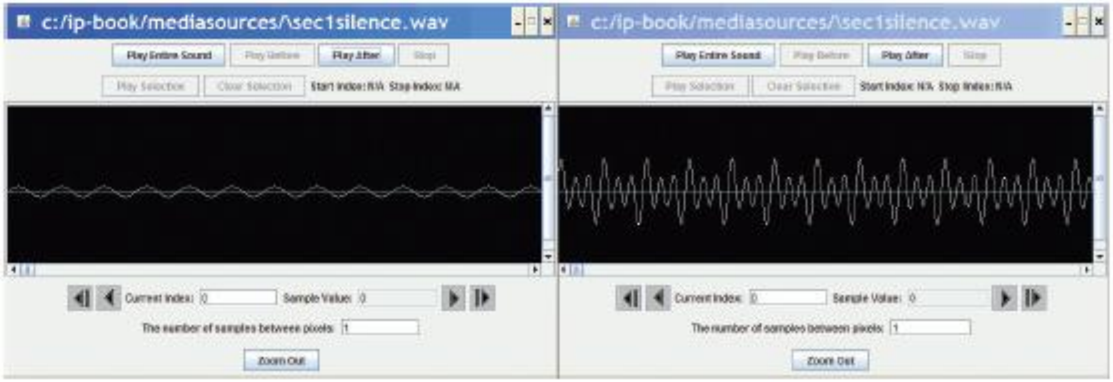
```
>>> f440=sineWave (440 ,2000, 1)
>>> f880=sineWave (880 ,4000, 1)
>>> f1320=sineWave (1320 ,8000, 1)
>>> addSoundInto(f880 ,f440)
>>> addSoundInto(f1320 ,f440)
>>> play(f440)
>>> explore(f440)
>>> just440=sineWave (440 ,2000, 1)
>>> play(just440)
>>> explore(f440)
```

Adding together 440Hz, 880Hz, and 1320Hz, with increasing amplitudes.

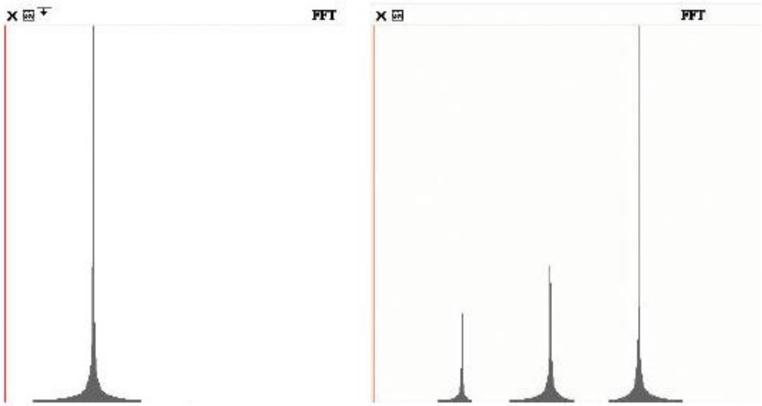
Comparing to a 440Hz wave

Comparing the waves

- Left, 440 Hz; Right, combined wave.



In Explorer



440, 880, 1320

In the Spectrum view in MediaTools

Subtractive synthesis

- The most widely used type of synthesis in electronic music production
- A subtractive synthesizer has **oscillators** which generate tones, and **filters** which shape the frequencies – both of which can be controlled by envelopes
  - Many classic synthesizers from the 1960s/1970s were based on **analogue** oscillators and filters
  - Modern synthesizers are **digital**, although many aim to model the characteristic sound of analogue components

A challenge (for the adventurous!)

- Incorporate subtractive synthesis (filtering) into your tinkering audio project
- Use the formulae here
  - http://www.musicdsp.org/files/Audio-EQ-Cookbook.txt
  - Don't be put off by the jargon – you don't need to understand the maths behind it to use it
  - See also http://blog.bjornroche.com/2012/08/basic-audio-eqs.html

Tinkering Audio

ENVELOPES

Envelopes

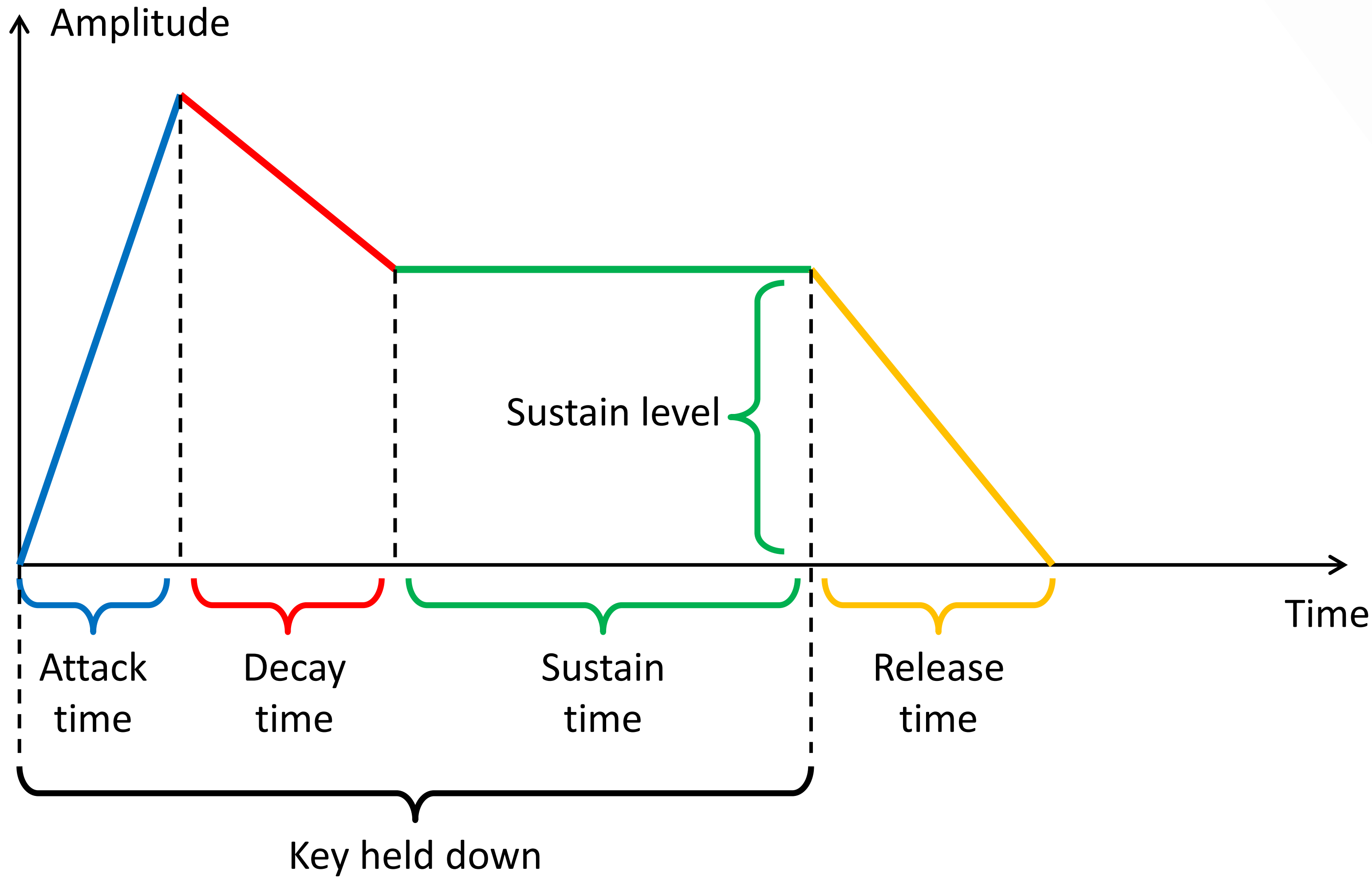
- Static sounds are not very interesting
- Almost all synthesizers allow the musician to manipulate **envelopes**
- Envelopes allow characteristics of the sound (volume, pitch, timbre etc) to vary over time

http://www.animations.physics.unsw.edu.au/jw/timbre-envelope.htm

ADSR envelopes

- The most common type of envelope is the **ADSR envelope**
  - **Peak level:** the peak amplitude that the note achieves
  - **Attack time**: how quickly the sound reaches peak level when a note is played
  - **Decay time**: how quickly the sound goes from peak amplitude to the sustain level after the initial peak
  - **Sustain level**: the amplitude that the sound maintains until the note stops playing
  - **Sustain time:** how long the sustain level is maintained (this can be a parameter, or can be determined by how long the musician holds the key down)
  - **Release time**: how quickly the note fades away when it stops playing

ADSR envelope



Exercise

- Adapt the sineWave function to create an attack-sustain-release envelope (add decay for a full ADSR envelope if you're feeling adventurous!)

- sineWave(frequency, volume, attack_time,

  sustain_time, release_time)

Tinkering Audio

PARSING A NOTATION FOR TONES

Parsing a Notation

Parsing a Notation: ABC Notation

```
X:1
T:The Legacy Jig
M:6/8
L:1/8
R:jig
K:G
GFG BAB | gfg gab | GFG BAB | d2A AFD |
GFG BAB | gfg gab | age edB |1 dBA AFD :|2 dBA ABd |:
 efe edB | dBA ABd | efe edB | gdB ABd |
efe edB | d2d def | gfe edB |1 dBA ABd :|2 dBA AFD |]
```

Parsing a Notation: ABC Notation

- Lines in the first part of the tune notation, beginning with a letter followed by a colon, indicate various aspects of the tune such as:
  - the index, when there are more than one tune in a file (X:),
  - the title (T:),
  - the time signature (M:),
  - the default note length (L:),
  - the type of tune (R:)
  - and the key (K:).
- Lines following the key designation represent the tune. This example can be translated into traditional music notation using one of the abc conversion tools.

Further Notes on ABC

- http://abcnotation.com/examples

A Simpler Parser – Key Concepts

- Reading notes and note durations from a string
- **Token** – a collection of symbols with a particular meaning (e.g., a note and its duration)
- **Delimiter** – the symbol that separates the tokens (e.g., a blank space – remember this is still encoded in the computer!)

Useful Python operations for parsing

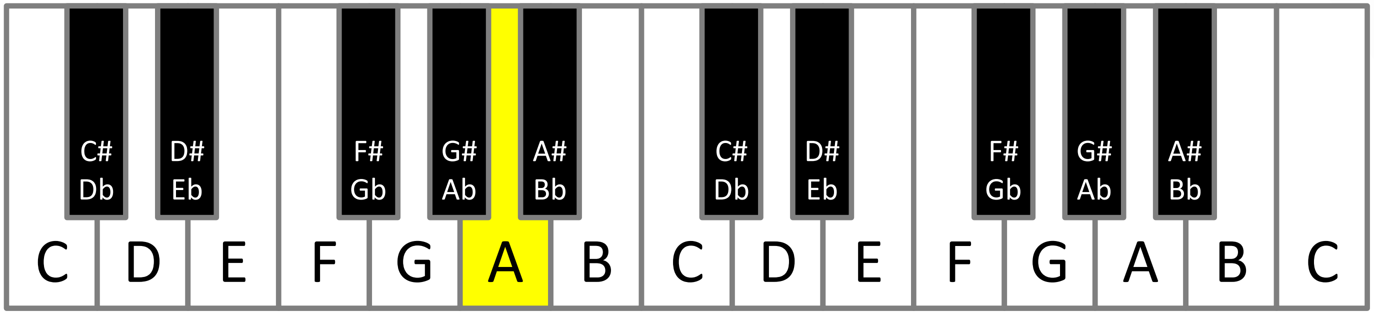
Splitting by delimiter

Getting a single character

Slicing

```
>>> s = "Hello world"

>>> s.split(' ')
['Hello', 'world']

>>> s[4]
'o'

>>> s[3:7]
'lo w'
>>> s[:3]
'Hel'
>>> s[7:]
'orld'
```

Converting notes to frequencies



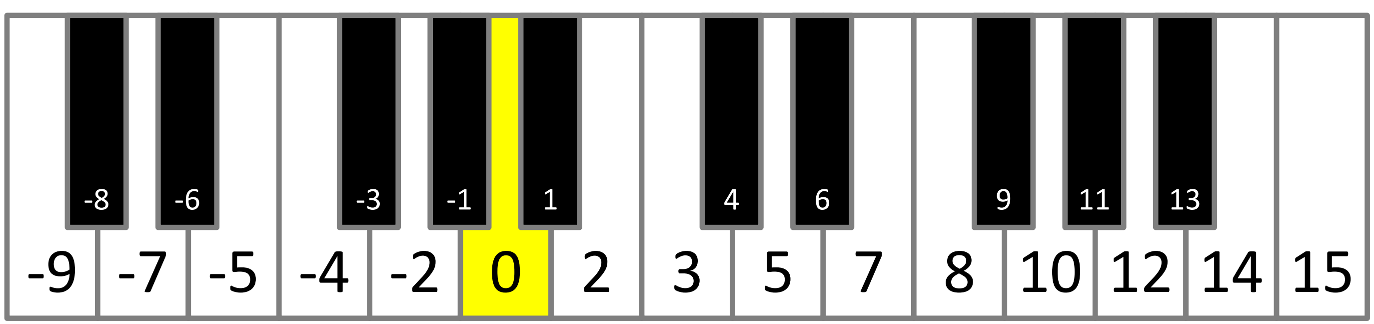
- The A above middle C on a piano is **440Hz**
- Notes **double** in frequency every **octave up**, and **halve** in frequency every **octave down**
  - A = 55Hz, 110Hz, 220Hz, 440Hz, 880Hz, 1760Hz, ...
- 1 **octave** = 12 **semitones**
  - A, A#, B, C, C#, D, D#, E, F, F#, G, G#, A

Converting notes to frequencies

- Use this formula: $$f = 440 \times 2^{\frac{n}{12}}$$

```
frequency = 440.0 * 2.0 ** (note_number / 12.0)
```

- note_number is the **offset** of the note, in semitones, from A=440Hz



Exercise

- Write a parser for your synthesizer to convert notes input as chars into actual sounds

- Use it to play a melody

Tinkering Audio

RANDOM NUMBER GENERATION

Adding new capabilities: Modules

- Sometimes we need to add capabilities to Python beyond those built into the basic interpreter
- We do this by **importing** external **modules**
- A module is a file with function and class definitions
- By **importing** the module, we make the module's capabilities available to our program
  - The module is loaded as if its contents had been typed into your program file

Python's Standard Library

- Python comes with an extensive **library** of modules
- Includes modules for mathematics, random number generation, file reading and writing, networking, operating system functions, parsing, data compression, cryptography, even interpreting Python code

Accessing Pieces of a Module

- We access the additional capabilities (functions, constants, classes) of a module using **dot notation**, after we **import** the module
- How do you know what's there?
  - Check the documentation: https://docs.python.org/2/library/index.html
  - Some IDEs (like PyCharm, but unfortunately not JES) have auto-completion and built-in documentation
  - There are a couple of books (Lundh, Hellmann) that describe the modules in more detail than the online documentation

An Interesting Module: Random

```
>>> import random
>>> for i in range(1,10):
...     print random.random()
...
0.82113693141939280
0.63542667797032460
0.9460060163520159
0.904615696559684
0.33500464463254187
0.08124982126940594
0.0711481376807015
0.7255217307346048
0.2920541211845866
```

Useful random functions

- **random.uniform(a, b):** get a random **floating point** number between a and b
- **random.randrange(a, b):** get a random **integer** between a and b-1
- **random.choice(list):** choose a random element from a list
- **random.shuffle(list):** randomise the order of a list
- **random.seed(x):** can be used to make your program generate the same sequence of random numbers each time it is run
- Many others – see the documentation for details

Randomly Choosing Notes from a List

```
>>> for i in range(1,5):
...        print random.choice([“C1", “D1", “E1",
              “F1", “G1", “A1", “B1",“C2",“D2“, “D3”])
...            print random.randrange(1, 8)
...
C1
4
E1
6
...
```

Exercise: Randomly Generating Melodies

- Given a list of notes and note durations, we can randomly take one from each to make a simple melody.

- Integrate a random number generator into the synthesizer that you have just created to randomly produce a simple melody.

White noise

```
import random

def makeNoise(amplitude, length):
  # Create a blank sound
  buildNoise = makeEmptySoundBySeconds(length)
  # Make some noise!
  for pos in range(getLength(buildNoise)):
    rawSample = random.uniform(-1, 1)
    sampleVal = int(amplitude * rawSample)
    setSampleValueAt(buildNoise, pos, sampleVal)
  return buildNoise
```

Tinkering Audio Assignment

- *Quality* **over** *quantity*
- **Avoid** feature creep and stick to *no more than* **<u>six</u>** algorithms for your assignment, such as this exemplar:

1st. tone generation
2nd. tone combination
3rd. audio splice and swap
4th. audio envelopes and echoes
5th. parsing tokens into audio
6th. random audio generation

- There can be overlap in groups, but some algorithms are better for sound effects (e.g. resampling) while others for melodies (e.g. parsing tokens into audio)