INFT1004 - SEMESTER 1 - 2017		LECTURE TOPICS	
Week 1	Feb 27	Introduction, Assignment, Arithmetic	
Week 2	Mar 6	Sequence, Quick Start, Programming Style	
Week 3	Mar 13	Pictures, Functions, Media Paths	
Week 4	Mar 20	Arrays, Pixels, For Loop, Reference Passing	
Week 5	Mar 27	Nested Loops, Selection, Advanced Pictures	
Week 6	Apr 3	Lists, Strings, Input & Output, Files	Practical Test
Week 7	Apr 10	Drawing Pictures, Program Design, While Loop	Assignment set
Recess	Apr 14 – Apr 23	Mid Semester Recess Break	
Week 8	Apr 24	No Lecture / Revision and Assignment in Labs	
Week 9	May 1	Data Structures, Processing sound	
Week 10	May 8	Advanced sound	Assignment part 1 due 8:00am Tue, May 9
Week 11	May 15	Movies, Scope, Import	
Week 12	May 22	Turtles, Writing Classes	Assignment part 2 due 8:00am Tue, May 23
Week 13	May 29	Revision	
Mid Year Examination Period - MUST be available normal & supplementary period			

Lecture Topics and Lab topics are the same for each week

INFT1004 - SEMESTER 1 - 2017		LECTURE TOPICS	
Week 1	Feb 27	Introduction, Assignment, Arithmetic	
Week 2	Mar 6	Sequence, Quick Start, Programming Style	
Week 3	Mar 13	Pictures, Functions, Media Paths	
Week 4	Mar 20	Arrays, Pixels, For Loop, Reference Passing	
Week 5	Mar 27	Nested Loops, Selection, Advanced Pictures	
Week 6	Apr 3	Lists, Strings, Input & Outpu	
Week 7	Apr 10	Drawing Pictures, Program D	
Recess	Apr 14 – Apr 23	Mid Semester Reces	
Week 8	Apr 24	No Lecture / Revision and A	
Week 9	May 1	Data Structures, Processing	
Week 10	May 8	Advanced sound	Assignment part 1 due 8:00am Tue, May 9
Week 11	May 15	Movies, Scope, Import	
Week 12	May 22	Turtles, Writing Classes	Assignment part 2 due 8:00am Tue, May 23
Week 13	May 29	Revision	
Mid Year Examination Period - MUST be available normal & supplementary period			

Lecture Topics and Lab topics are the same for each week

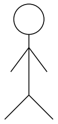
Assignment due next Tuesday at 8:00 am

INFT1004

Visual Programming

Module 9.1
Data Structures
(More Lists and Parallel Lists)

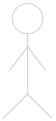
Many variables for the same thing



If we were collecting physical data about a person, we might have variables such as..

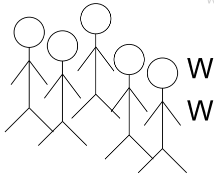
```
name
height
weight
```

Many variables for the same thing



If we were collecting physical data about a person, we might have variables such as..

```
name
height
weight
```

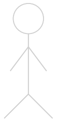


What if we had 5 people?

We could try....

```
name1,   name2,   name3,   name4,   name5,
height1, height2, height3, height4, height5,
weight1, weight2, weight3, weight4, weight5
```

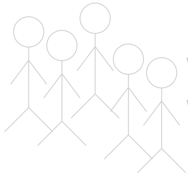
Many variables for the same thing



If we were collecting physical data about a person, we might have variables such as..

```
name
height
weight
```

But this is starting to look a little ridiculous.

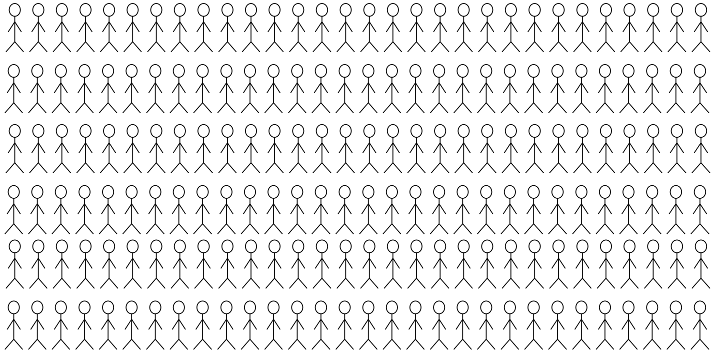


What if we had 5 people?

We could try....

```
name1,   name2,   name3,   name4,   name5,
height1, height2, height3, height4, height5,
weight1, weight2, weight3, weight4, weight5
```

Many variables for the same thing



Now what if we had 180 people? 5000 people?

There must be a better way.

Lists / Arrays

The array is a neat programming device for storing many values of the same type in a single variable with just one name.

In python arrays are implemented as lists

A list is similar to an array (in other languages) – although lists don't need to have elements all of the same type

We have already used lists in python for storing groups of pixels (and strings work a bit the same)

Iteration and Lists

Lists are a very useful data structure and work well with iteration (for loops and while loops)

We have already seen this with pictures and other lists.

Let’s look at some more examples and introduce the use of parallel arrays

Lists are also sequences – they are iterable

An example List

height	1.6	1.7	2.1	1.9	1.7
	`height[0]`	`height[1]`	`height[2]`	`height[3]`	`height[4]`

A list called height might have 5 values called `height[0], height[1], height[2], height[3],` and `height[4].`

height List

They're all called `height`, but each value has a different index – the number after it in parentheses.

`height`	1.6	1.7	2.1	1.9	1.7
index	0	1	2	3	4

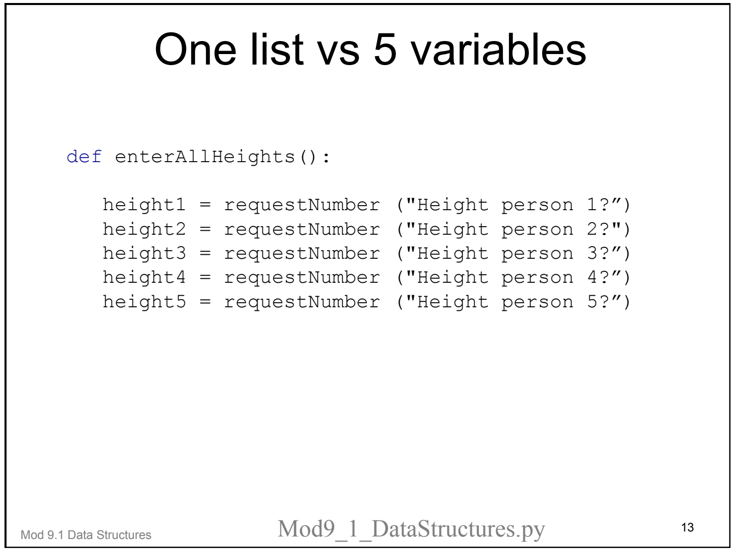
This is a list with 5 heights. We could just as easily have a list with 500 or 5000 heights.

Lists

They're all called dHeight, but each value has a different index – the number after it in parentheses.

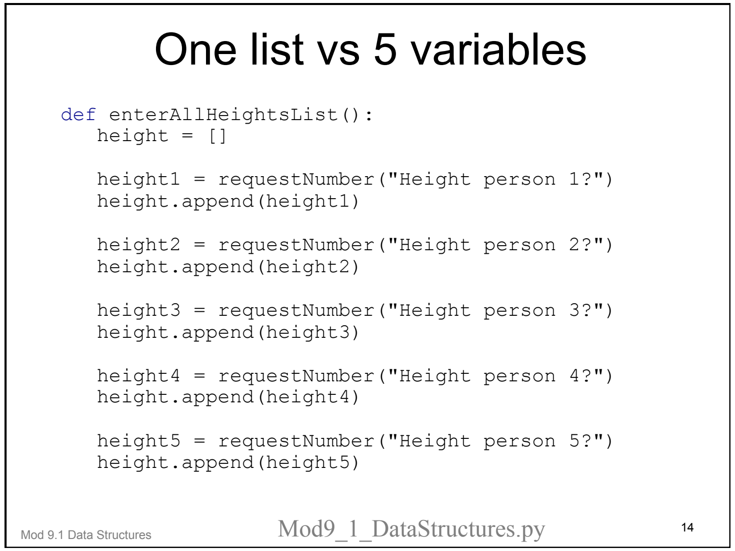
`height`	1.6	1.7	2.1	1.9	1.7
index	0	1	2	3	4

So what are the benefits of an list as against a number of distinct variables?



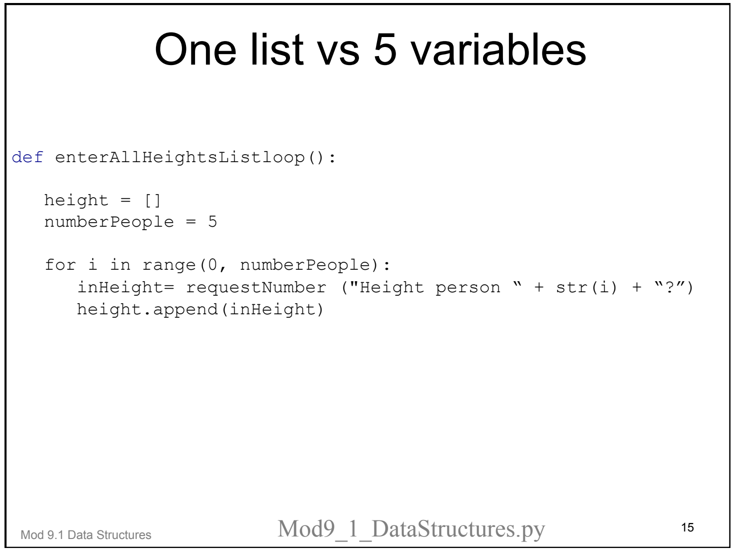
One list vs 5 variables

```
def enterAllHeights():

   height1 = requestNumber ("Height person 1?”)
   height2 = requestNumber ("Height person 2?")
   height3 = requestNumber ("Height person 3?”)
   height4 = requestNumber ("Height person 4?”)
   height5 = requestNumber ("Height person 5?”)
```



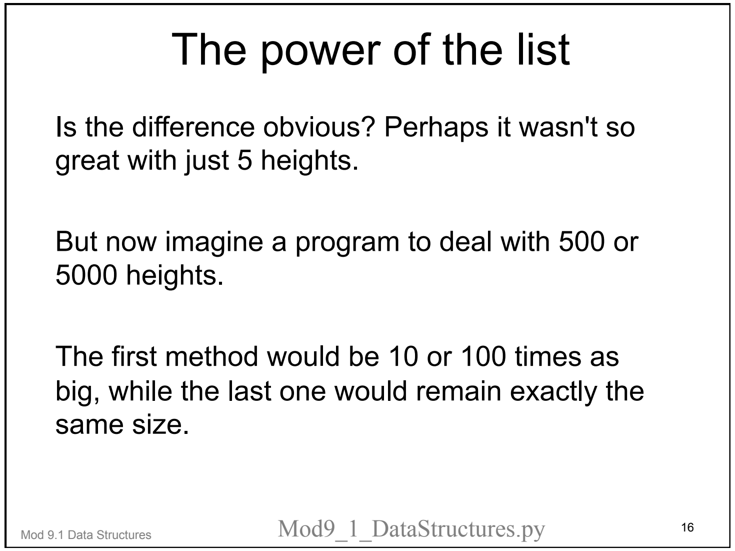
One list vs 5 variables

```
def enterAllHeightsList():
   height = []

   height1 = requestNumber("Height person 1?")
   height.append(height1)

   height2 = requestNumber("Height person 2?")
   height.append(height2)

   height3 = requestNumber("Height person 3?")
   height.append(height3)

   height4 = requestNumber("Height person 4?")
   height.append(height4)

   height5 = requestNumber("Height person 5?")
   height.append(height5)
```



One list vs 5 variables

```
def enterAllHeightsListloop():

   height = []
   numberPeople = 5

   for i in range(0, numberPeople):
      inHeight= requestNumber ("Height person “ + str(i) + “?”)
      height.append(inHeight)
```



The power of the list

Is the difference obvious? Perhaps it wasn't so great with just 5 heights.

But now imagine a program to deal with 500 or 5000 heights.

The first method would be 10 or 100 times as big, while the last one would remain exactly the same size.

The power of the list

The power of the array lies in the programmer's ability to use a variable as its index . . .

`height`	1.6	1.7	2.1	1.9	1.7
index	0	1	2	3	4
	↑ i				

. . . and thus to use loops to process each element in turn with the same small piece of code.

Index

The index of the first element is always zero . .

```
print(height[0])  #first element
```

So the index of the last element is one less than the number of elements.

```
lastIndex = len(height) -1
print(height[lastIndex])    #last element
```

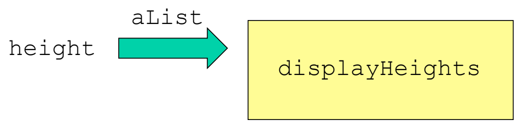
Lists – common errors

Missing the first element forgetting to start at 0
`height[0]`

Going over the end of the list – out of range
`height[len(height)]`

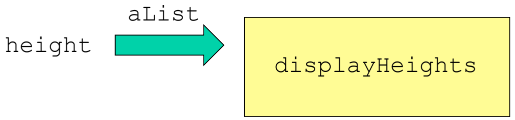
Lists as parameters

It makes sense to sometimes use a list as a parameter, and to pass it in as an argument when calling the method.



Lists as parameters

It makes sense to sometimes use a list as a parameter, and to pass it in as an argument when calling the method.



```
def displayList(aList):
   for i in range(0, len(aList)):
      print("list[" + str(i) + "]=" + str(aList[i]))

>>> displayList(height)
```

Parallel Lists

If we store people's names in one list,
their heights in a second list,
their weights in a third list. . .

names	"Keith"	"John"	"Mary"	"Jo"	"Sue"
height	1.6	1.7	2.1	1.9	1.7
weight	87	89	62	91	73
index	0	1	2	3	4

Parallel Lists

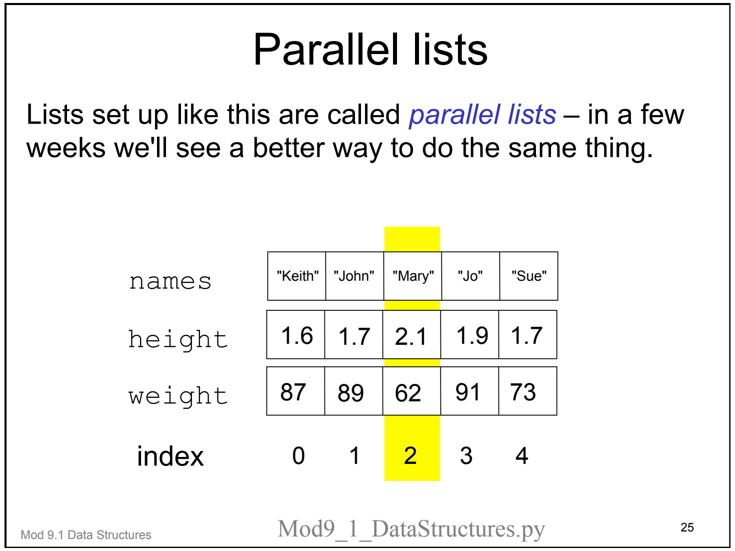
. . . so long as we don't shuffle the elements in any list, a particular index value refers to the same person in each list.

names	"Keith"	"John"	"Mary"	"Jo"	"Sue"
height	1.6	1.7	2.1	1.9	1.7
weight	87	89	62	91	73
index	0	1	2	3	4

Parallel lists

The person whose name is in `names[2]`
has the height that's in `height[2]`
and the weight that's in `weight[2]`

names	"Keith"	"John"	"Mary"	"Jo"	"Sue"
height	1.6	1.7	2.1	1.9	1.7
weight	87	89	62	91	73
index	0	1	2	3	4

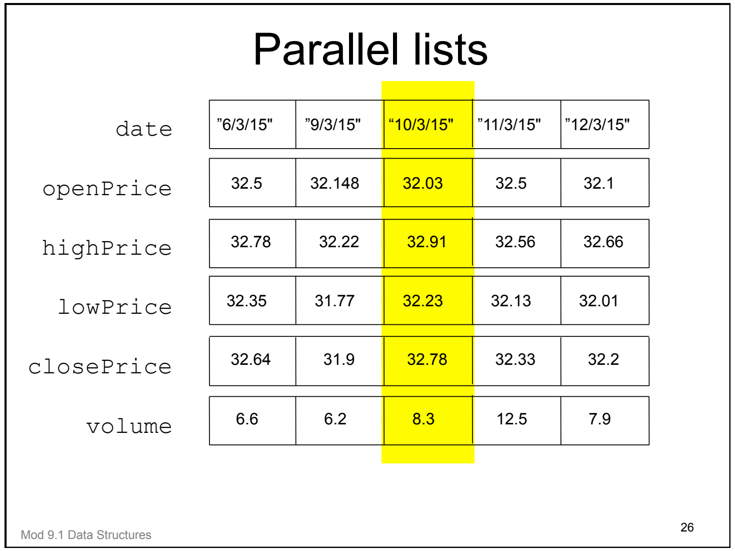


Parallel lists

Lists set up like this are called *parallel lists* – in a few weeks we'll see a better way to do the same thing.

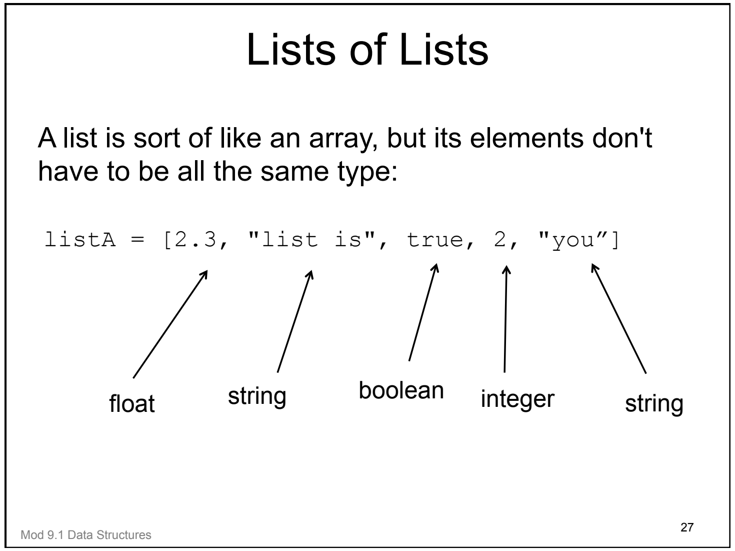
names	"Keith"	"John"	"Mary"	"Jo"	"Sue"
height	1.6	1.7	2.1	1.9	1.7
weight	87	89	62	91	73
index	0	1	2	3	4

Mod9_1_DataStructures.py



Parallel lists

date	”6/3/15"	”9/3/15"	“10/3/15"	”11/3/15"	”12/3/15"
openPrice	32.5	32.148	32.03	32.5	32.1
highPrice	32.78	32.22	32.91	32.56	32.66
lowPrice	32.35	31.77	32.23	32.13	32.01
closePrice	32.64	31.9	32.78	32.33	32.2
volume	6.6	6.2	8.3	12.5	7.9



Lists of Lists

A list is sort of like an array, but its elements don't have to be all the same type:

```
listA = [2.3, "list is", true, 2, "you”]
```

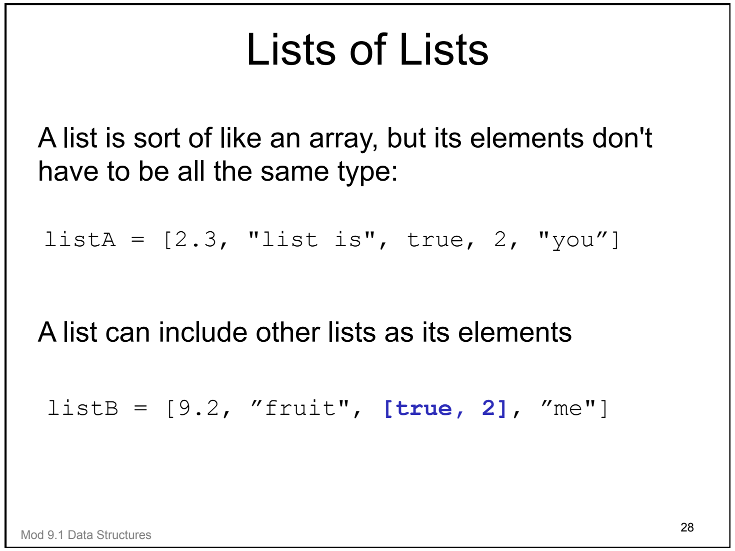
float

string

boolean

integer

string



Lists of Lists

A list is sort of like an array, but its elements don't have to be all the same type:

```
listA = [2.3, "list is", true, 2, "you”]
```

A list can include other lists as its elements

```
listB = [9.2, ”fruit", [true, 2], ”me"]
```

Lists – different elements

`listB[2]` is the list `[true, 2]`

```
listB = [9.2, ”fruit", [true, 2], ”me"]
```

`listB[2][0]` is the boolean `true`

```
listB = [9.2, ”fruit", [true, 2], ”me"]
```

Lists – different elements

Lists with sublists can represent complex structures:

```
listC = ["Food groups", ["protein", ["meat",
"fish", "egg", "soy"]], ["carbohydrate", ["sugar",
"starch"]], ["fat", ["oil", "lard", "butter"]],
["alcohol", ["beer", "wine", "spirits"]]]
```

What is listC[4][1][0]?

Lists – a new concept

Lists with sublists can represent complex structures:

```
listC = ["Food groups", ["protein", ["meat",
"fish", "egg", "soy"]], ["carbohydrate", ["sugar",
"starch"]], ["fat", ["oil", "lard", "butter"]],
["alcohol", ["beer", "wine", "spirits"]]]
```

What is listC[4][1][0]?

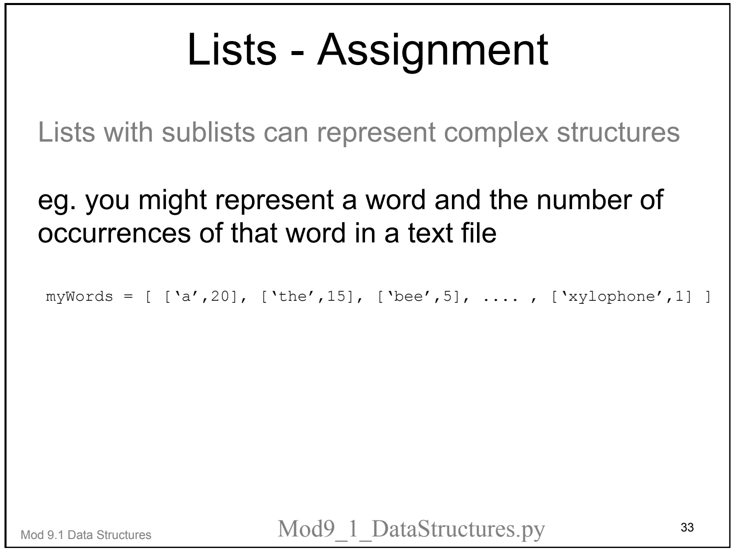
Lists

Lists with sublists can represent complex structures

eg you might represent an image as a list of row lists that contains a list of three integers (colour channels)

```
myPicture3By3 = [
  [ [120, 230, 150], [32,33,120], [190, 180, 20]],
  [ [110, 130,  50], [34,37,120], [195, 170, 30]],
  [ [180, 230,  50], [28,43,120], [196, 183, 40]] ]
```

3 rows of 3 columns with a list of 3 colour channel (r,g,b) values



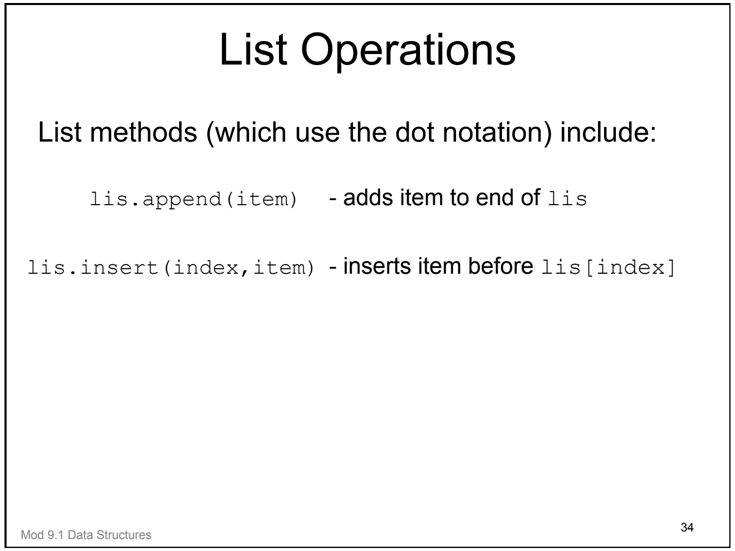
Lists - Assignment

Lists with sublists can represent complex structures

eg. you might represent a word and the number of occurrences of that word in a text file

```
myWords = [ [‘a’,20], [‘the’,15], [‘bee’,5], .... , [‘xylophone’,1] ]
```

Mod9_1_DataStructures.py

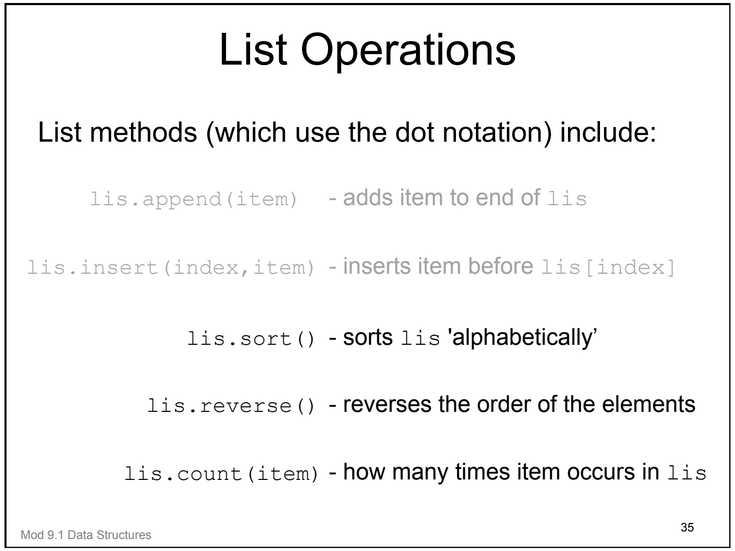


List Operations

List methods (which use the dot notation) include:

`lis.append(item)` - adds item to end of `lis`

`lis.insert(index,item)` - inserts item before `lis[index]`



List Operations

List methods (which use the dot notation) include:

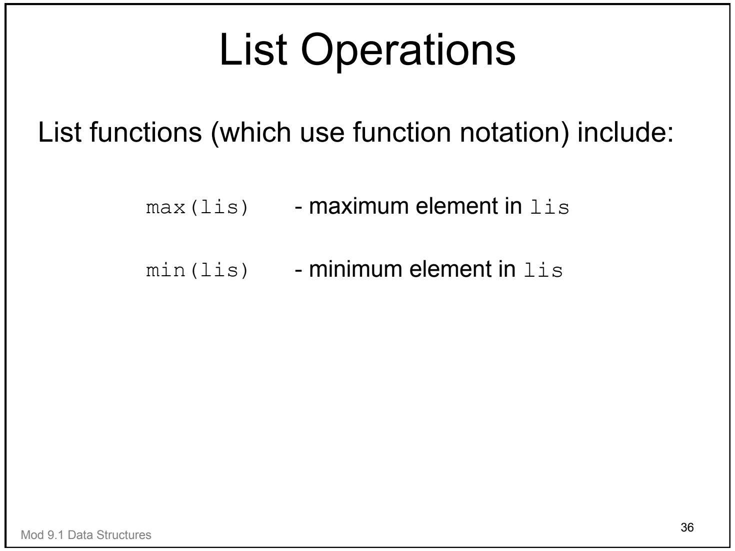
`lis.append(item)` - adds item to end of `lis`

`lis.insert(index,item)` - inserts item before `lis[index]`

`lis.sort()` - sorts `lis` 'alphabetically’

`lis.reverse()` - reverses the order of the elements

`lis.count(item)` - how many times item occurs in `lis`



List Operations

List functions (which use function notation) include:

`max(lis)` - maximum element in `lis`

`min(lis)` - minimum element in `lis`

Making a list of words

`split()` is a not a method of lists, but of strings

– but it does produce a list

When using it, you specify a separator

It produces a list of strings as separated by that separator

Making a list of words

```
url = "http://www.newcastle.edu.au/profiles”

url.split(".")
```

will give the list

```
['http://www', 'newcastle', 'edu', 'au/profiles']
```

~~Parallel~~ Lists??

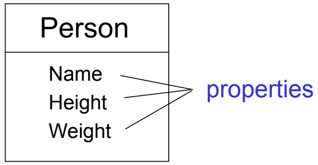
Actually there is a much better way to solve this problem - use a class!

names	"Keith"	"John"	"Mary"	"Jo"	"Sue"
height	1.6	1.7	2.1	1.9	1.7
weight	87	89	62	91	73
index	0	1	2	3	4

Parallel Lists??

You could create your own Person class.

You could give it the properties you need



Person

Name

Height

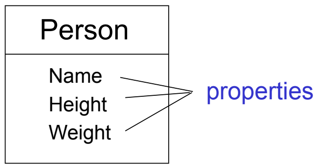
Weight

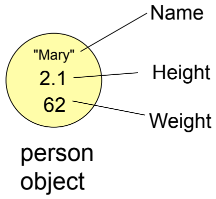
properties

Parallel Lists??

You could create your own Person class.

You could give it the properties you need

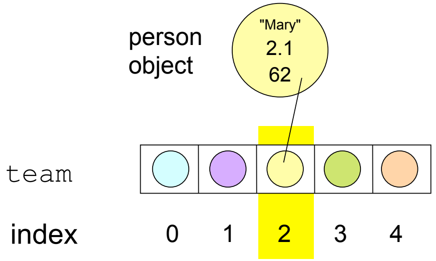




Then you can create as many person objects as you need.

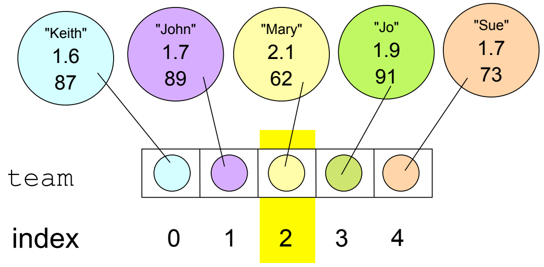
A Person Class

Then you can have an array of these person objects.



A Person Class

Note how nicely encapsulated the data for each person is. No chance of getting data mixed up like there is with parallel arrays



A Person Class

Of course I have left out a few details..
You need to be able to define your class in python

A Person Class

Of course I have left out a few details..
You need to be able to define your class in Python

- specify the attributes in your class (type and name)
- write some methods that allow people to use these attributes (get & set)

A Person Class

Of course I have left out a few details..
You need to be able to define your class in Python

- specify the attributes in your class (type and name)
- write some methods that allow people to use these attributes (get & set)
- write some useful methods in your class
- write some special methods to instantiate your class (constructers)

A Person Class

You will also need to know how to use your classes.

You will need to be able to declare, instantiate and initialise objects of your class.

You will need to be able to use the attributes and methods provided by your class.

A Person Class

You will also need to know how to use your classes.

You will need to be able to declare, instantiate and initialise objects of your class.

You will need to be able to use the attributes and methods provided by your class.

(Actually this is all no different then using any other class.) (More later)

INFT1004

Visual Programming

Module 9.2
Introduction to Sound

Guzdial & Ericson - Third Edition - chapters 6 and 7
Guzdial & Ericson - Fourth (Global) Edition – chapters 7 and 8

Working with Pictures

We see a picture as continuous patches of colour.

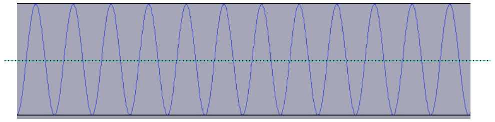
But a digitised picture is broken into individual ‘pixels’, each representing the colour value at one small point

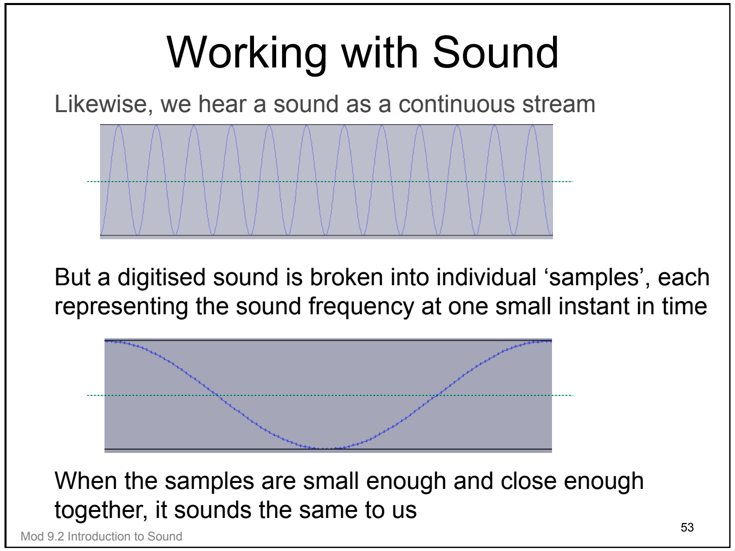
Working with Pictures

When the pixels are small enough and close enough together, it looks the same to us.

Working with Sound

Likewise, we hear a sound as a continuous stream



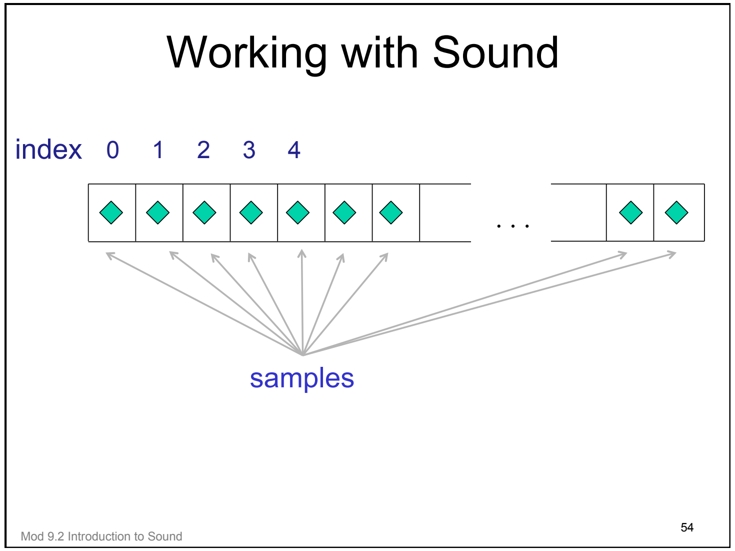


Working with Sound

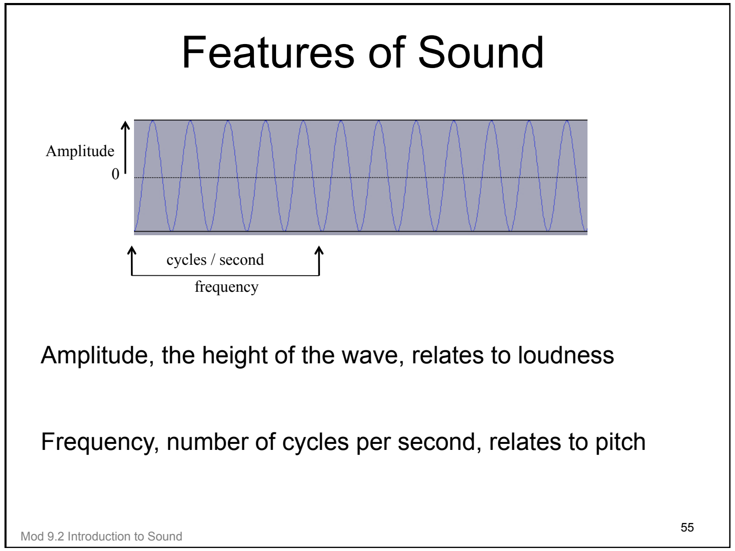
Likewise, we hear a sound as a continuous stream

But a digitised sound is broken into individual ‘samples’, each representing the sound frequency at one small instant in time

When the samples are small enough and close enough together, it sounds the same to us



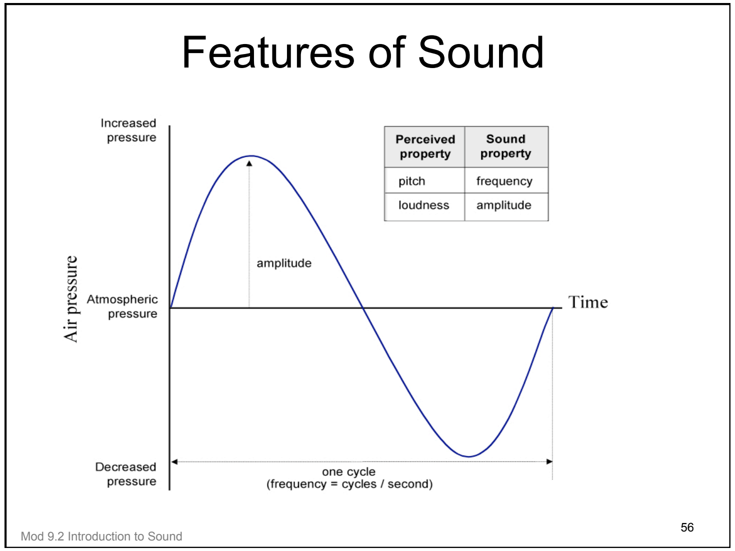
Working with Sound



Features of Sound

Amplitude, the height of the wave, relates to loudness

Frequency, number of cycles per second, relates to pitch



Features of Sound

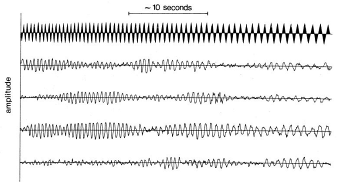
Perceived property	Sound property
pitch	frequency
loudness	amplitude

Features of Sound

Overtones are additional frequencies that turn pure sound into rich sound

Real waves have different shapes (sine, square, triangle, indeterminate)

Very few real sounds are pure in pitch or wave shape



Features of Sound

Sounds also have a quality called Timbre

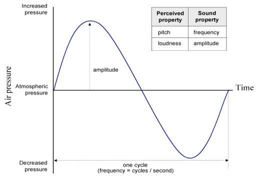
This is a quality of a sound that make them sound different (even though they have the same amplitude and frequency

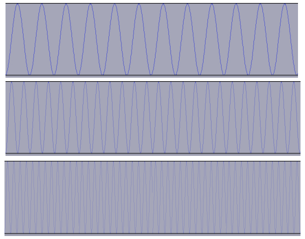
(e.g. flute compared to a violin, different voices)

Features of Sound

Loudness and pitch both relate *logarithmically* to amplitude and frequency:

Doubling the amplitude – increases loudness by same amount
Doubling the frequency – increases pitch by same amount





Features of Sound

Loudness and pitch both relate *logarithmically* to amplitude and frequency:

Doubling the amplitude – increases loudness by same amount
Doubling the frequency – increases pitch by same amount

Actually it's more complex than this - we respond differently to low and high frequencies at low amplitudes (loudness button) - and responses to different frequencies change with age.

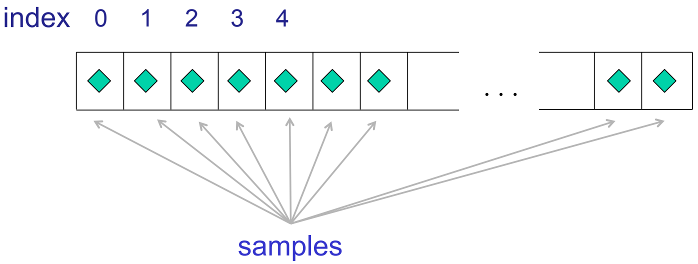
Sampling rates and Nyquist

Nyquist theorem: for a reasonable recording, sample at twice the rate of the highest frequency in the sample

Human speech goes to 4,000Hz (cycles per second), so for a good speech recording we need to sample at 8,000Hz (8000 samples per second)

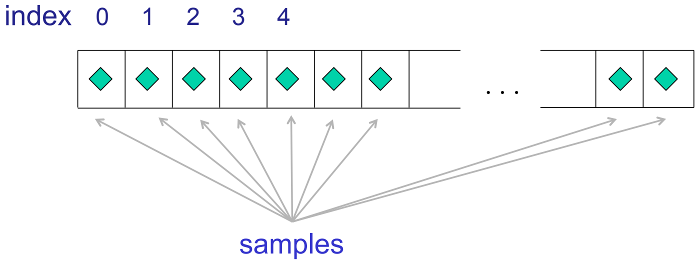
Human hearing goes to 22,000Hz, so for a good music recording we need to sample at 44,000Hz (44,000 samples per second)

A Digital Sound has samples



Each sample has an amplitude value

Each Sample has an Amplitude



Each sample has an amplitude value

Possible Amplitude values

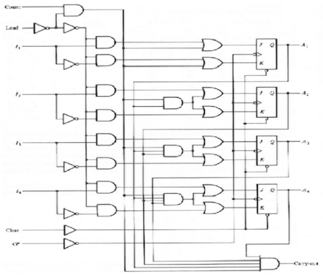
-32, 768 ----- 32, 767

EXAMPLE - Up the volume

```
def upVolume(aSound):

   # This function doubles the amplitude
   # of a sound
   # Note: if multiplied sample values
   #       exceed 32767 they will be clipped

   for sample in getSamples(aSound):
      value = getSampleValue(sample) * 2
      setSampleValue(sample, value)
```

Mod9_2_IntroductionSound.py

Binary numbers



Aside – computers use binary, off/on representing 0/1

off ➔ 0
on ➔ 1

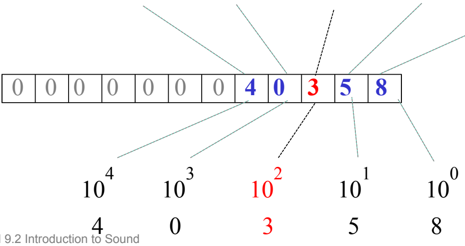
Decimal and Binary numbers

In the decimal system there are 10 digits, from 0 to 10 – 1

Column n (from the right) gives the number of times 10^{n-1} is found in the number

column 3

$$40358 = 4 \text{ x } 10^4 + 0 \text{ x } 10^3 + 3 \text{ x } 10^2 + 5 \text{ x } 10^1 + 8 \text{ x } 10^0$$



Decimal

base 10

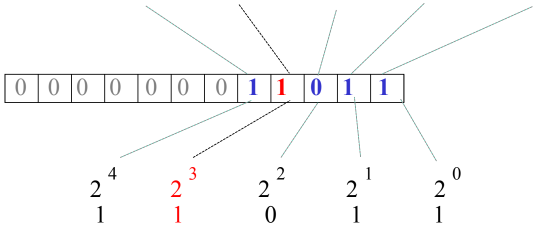
Decimal and Binary numbers

In the binary system there are 2 digits, 0 and 1

Column n (from the right) gives the number of times 2^{n-1} is found in the number

column 4

$$11011 = 1 \text{ x } 2^4 + 1 \text{ x } 2^3 + 0 \text{ x } 2^2 + 1 \text{ x } 2^1 + 1 \text{ x } 2^0$$



Binary

base 2

(27 in base 10)

Decimal and Binary numbers

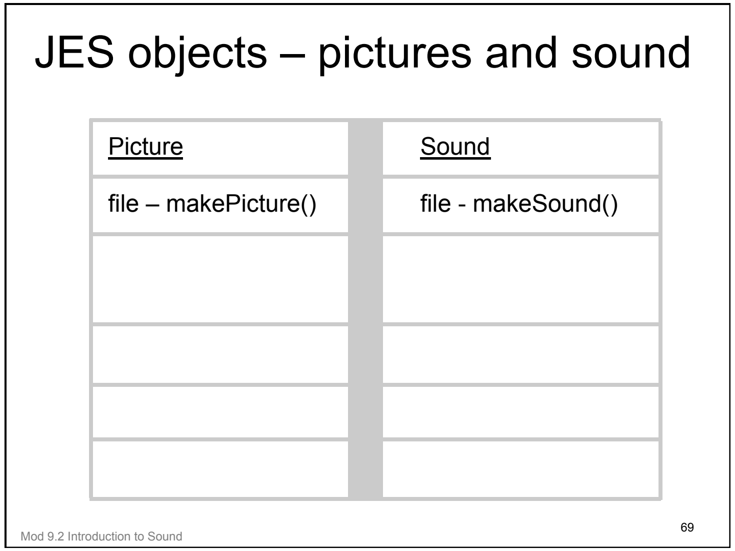
For sound we normally use 2 bytes (16 bits) to store the sound amplitude.

One bit stores positive or negative

The remaining 15 bits allows for numbers between 0 and 2^{15}

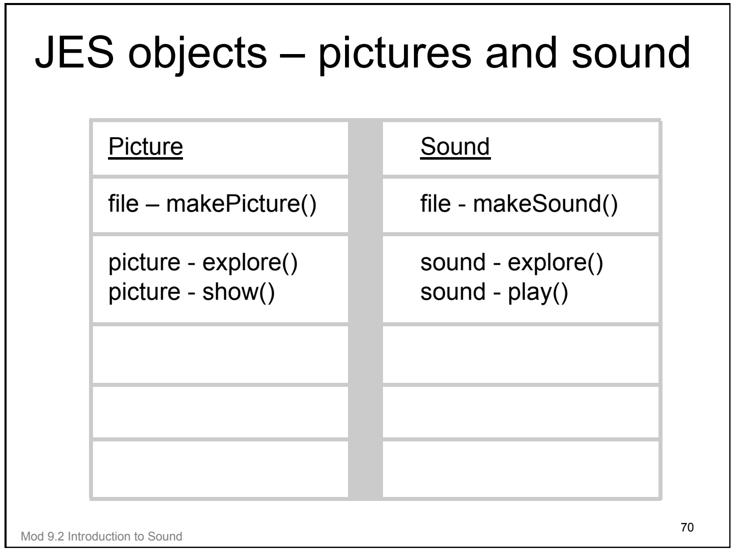
Possible values

-32, 768 ----- 32, 767



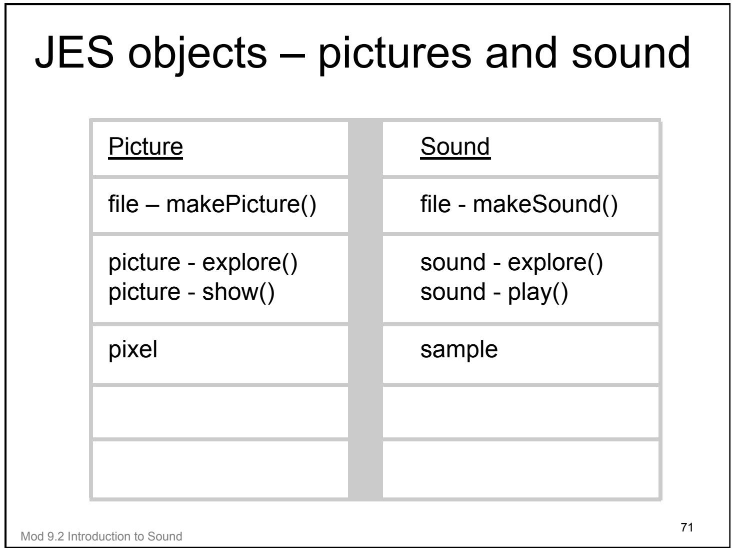
JES objects – pictures and sound

Picture	Sound
file – makePicture()	file - makeSound()



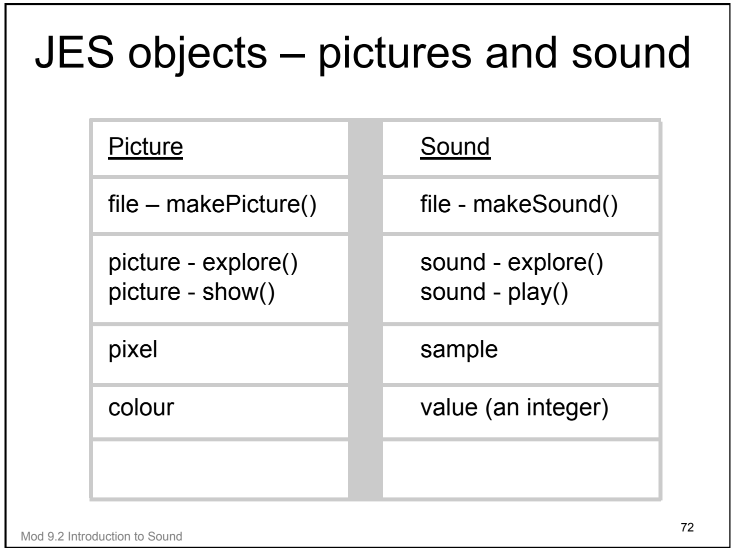
JES objects – pictures and sound

Picture	Sound
file – makePicture()	file - makeSound()
picture - explore() picture - show()	sound - explore() sound - play()



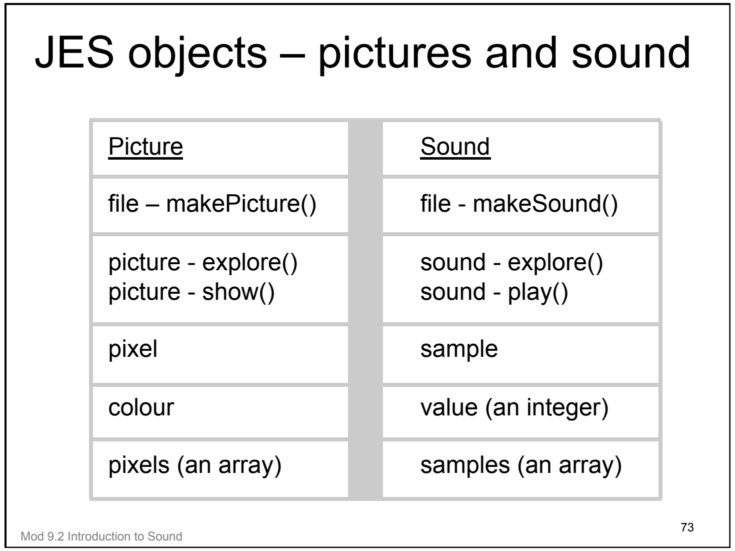
JES objects – pictures and sound

Picture	Sound
file – makePicture()	file - makeSound()
picture - explore() picture - show()	sound - explore() sound - play()
pixel	sample



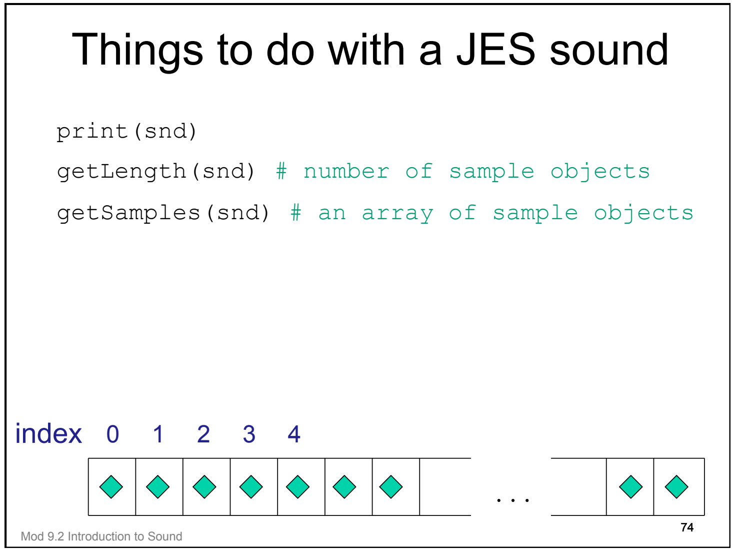
JES objects – pictures and sound

Picture	Sound
file – makePicture()	file - makeSound()
picture - explore() picture - show()	sound - explore() sound - play()
pixel	sample
colour	value (an integer)



JES objects – pictures and sound

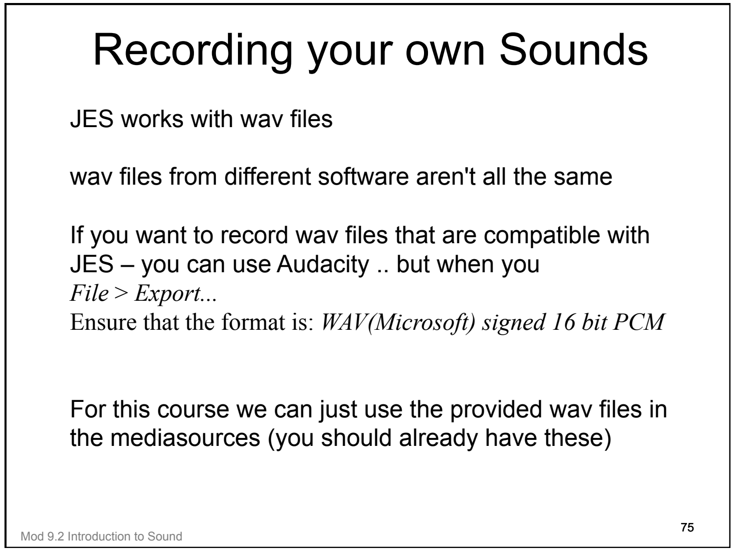
Picture	Sound
file – makePicture()	file - makeSound()
picture - explore() picture - show()	sound - explore() sound - play()
pixel	sample
colour	value (an integer)
pixels (an array)	samples (an array)



Things to do with a JES sound

```
print(snd)
getLength(snd) # number of sample objects
getSamples(snd) # an array of sample objects
```

index 0 1 2 3 4
. . .



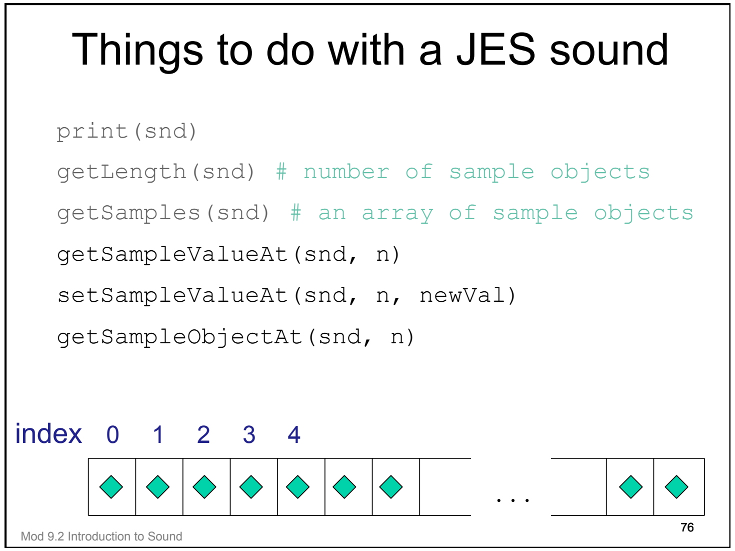
Recording your own Sounds

JES works with wav files

wav files from different software aren't all the same

If you want to record wav files that are compatible with JES – you can use Audacity .. but when you
File > Export...
Ensure that the format is: *WAV(Microsoft) signed 16 bit PCM*

For this course we can just use the provided wav files in the mediasources (you should already have these)



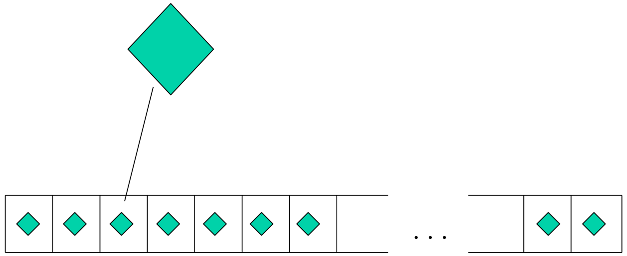
Things to do with a JES sound

```
print(snd)
getLength(snd) # number of sample objects
getSamples(snd) # an array of sample objects
getSampleValueAt(snd, n)
setSampleValueAt(snd, n, newVal)
getSampleObjectAt(snd, n)
```

index 0 1 2 3 4
. . .

Things to do with a sample object

```
getSampleValue(samp)

setSampleValue(samp, newVal)
```



More things to do with a JES sound

```
writeSoundTo(snd, file)

play(snd)

blockingPlay(snd)

explore(snd)
```

Processing Samples in an Array

Every element of a sample array is a sample object.

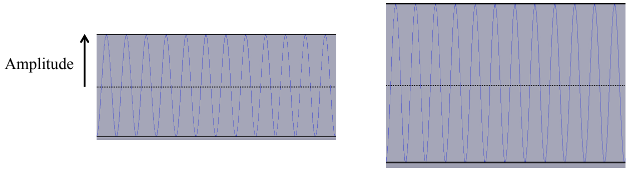
Once you've got the array of samples you can access samples ..

```
soundArray = getSamples(aSound)
value = soundArray[i]
```

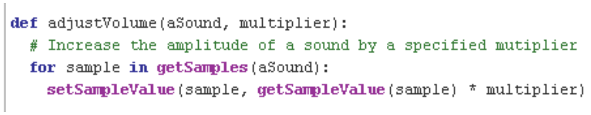
This might be easier than processing samples like this..

```
value = getSampleValueObjectAt(aSound, i)
```

Increasing & decreasing amplitude



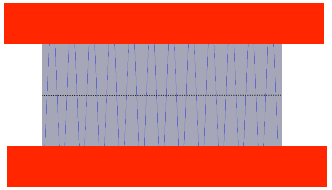
Once we've made a sound from a file, it's easy to adjust the amplitude (and thus the volume)



```
def adjustVolume(aSound, multiplier):
  # Increase the amplitude of a sound by a specified mutiplier
  for sample in getSamples(aSound):
    setSampleValue(sample, getSampleValue(sample) * multiplier)
```

Mod9_2_IntroductionSound.py

Increasing & decreasing amplitude

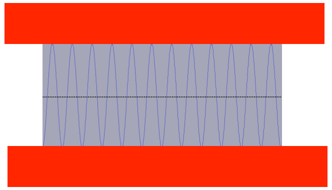


Note that if we adjust too far, we get ‘clipping’ – the biggest (positive and negative) values are chopped off

And we can’t reverse it – even if we reduce the amplitude the sound remains clipped.

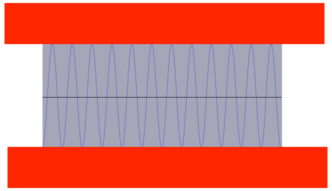
Normalising sound

Normalising sound means increasing the amplitude just as far as we can without clipping



Normalising sound

Normalising sound means increasing the amplitude just as far as we can without clipping



1. We need to find the biggest amplitude (positive or negative)
2. Increase that up to the maximum possible (32767)
3. Multiply every other sample by the same amount

Normalising sound

Read the book and the module code very carefully to understand how it works

```
def normalise(aSound):
# This function increases the amplitude of aSound as much
# as possible without clipping

# 1. First find the largest Amplitude sample in the sound

# 2. Work out the biggest amount you can scale this value up.
#    This scaling factor is the largest value you can multiply
#    the biggest amplitude sample by without clipping it.

# 3. Scale up all samples by the scaling factor
```

Mod9_2_IntroductionSound.py

Working in specific ranges

Just as with pictures, we can work in particular ranges of the array rather than the whole sound

Just as with pictures, we use explore() to find the start and end of the range we're interested in (drag to make a selection, then play it)

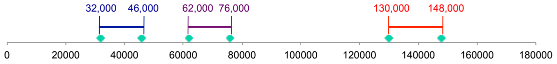
Working in specific ranges

Let's say we have a sound in which we want to increase the volume of three particular parts

First we use `explore()` to find their start and finish values

In a sound of 180,000 samples, we might want to boost the amplitude between

- 32,000 and 46,000
- 62,000 and 76,000
- 130,000 and 148,000



Increase amplitude in a range

We write a function that increases the amplitude by a specified amount in a single range (note use of array)

```
def adjustAmplitude(aSound, multiplier, startIndex, endIndex):
  # Adjust the amplitude of aSound by multipler in the samples
  # from startIndex to endIndex

  # First we need the array
  samples = getSamples(aSound)

  # Now we can select the samples that need adjusting and adjust them
  for index in range(startIndex, endIndex):
     setSampleValue(samples[index], multiplier * getSampleValue(samples[index]))
```

Mod9_2_IntroductionSound.py

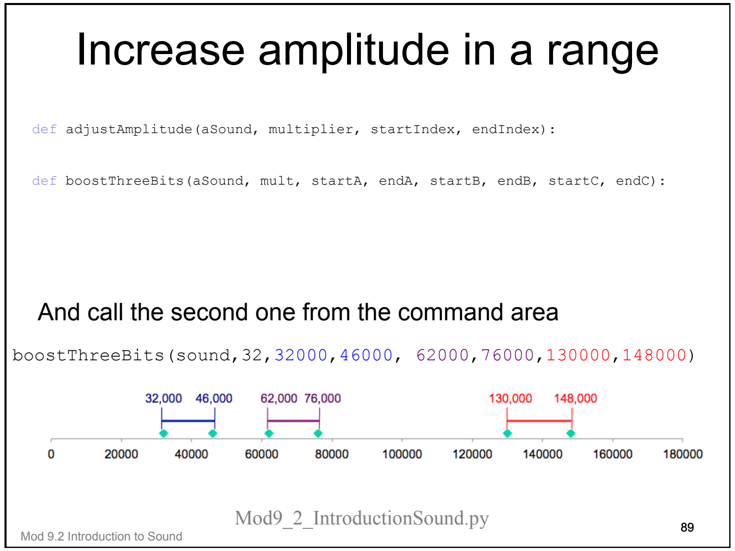
Increase amplitude in a range

```
def adjustAmplitude(aSound, multiplier, startIndex, endIndex):
```

Then we write another function that calls that function three times

```
def boostThreeBits(aSound, factor, startA, endA, startB, endB, startC, endC):
  # A highly specific function to boost the amplitude by factor just in
  # three specific ranges
  adjustAmplitude(aSound, factor, startA, endA)
  adjustAmplitude(aSound, factor, startB, endB)
  adjustAmplitude(aSound, factor, startC, endC)
```

Mod9_2_IntroductionSound.py



Increase amplitude in a range

```
def adjustAmplitude(aSound, multiplier, startIndex, endIndex):


def boostThreeBits(aSound, mult, startA, endA, startB, endB, startC, endC):
```

And call the second one from the command area

```
boostThreeBits(sound,32,32000,46000, 62000,76000,130000,148000)
```

Mod9_2_IntroductionSound.py