What is JythonMusic?

- JythonMusic is an open source environment for music making and creative programming activities.
- It is intended for musicians and programmers of all levels and backgrounds.
- JythonMusic is written in Python (gentle for beginners, powerful enough for experts).
- It provides libraries for music making, image manipulation, building graphical user interfaces (GUIs), and for connecting to external MIDI and OSC devices, such as digital pianos, smartphones, and tablets.

Why Python?

- Traditional languages (C++, Java) evolved for large-scale programming.
  - Emphasis on structure and discipline
  - Simple problems != simple programs
- Scripting languages (Perl, Python, TCL) designed for simplicity and flexibility.
  - Simple problems = simple, elegant solutions
  - More amenable to experimentation and incremental development
- Python: Ideal first language, useful throughout curriculum

(Quoted from John Zelle, “Teaching Computer Science with Python”, SIGCSE 2003 Workshop)
Courses

- JythonMusic has been used in
  - HONS 381 - “Computer Music on a Laptop: Composing, Performing, Interacting”

CS Principles - 7 Big Ideas

- Creativity - computing is a creative human activity.
- Abstraction - helps understand and solve problems.
- Data and Information - facilitate the creation of knowledge.
- Algorithms - tools for developing and expressing solutions.
- Programming - creative process that produces computational artifacts.
- The Internet - interconnecting devices provide ways to solve problems.
- Global Impact - computing enables innovation in other fields including mathematics, science, humanities, and arts.

Motivation - Laptop Orchestra

Image Sonification

bit.ly/CharlestonLaptopOrchestra

https://vimeo.com/64110119
Art Installation - Time Jitters

Hyperinstrument #1 - Monterey Mirror

Hyperinstrument #2 - SoundMorpheus

Introduction

• JythonMusic libraries
  • Music - primitives for creating music notes, phrases, parts, and scores, for playing live, and for reading and writing as MIDI or XML files
  • also primitives for loading and looping audio files
Introduction

• JythonMusic libraries (cont’d)
  • GUI - primitives for building graphical user interfaces for music performance and other interactive tasks
  • MIDI - primitives for connecting to external MIDI devices (e.g., pianos, guitars, synthesizers, etc.)
  • OSC - primitives for connecting to other devices via Open Sound Control (e.g., smartphones, tablets, computers, synthesizers, etc.)

• Image - primitives for supporting image sonification and other types of image manipulation
• Timer - primitives for scheduling events, such as specifying envelope manipulation of audio, as well as constructing various animation tasks
• Zipf - primitives for extracting measurements from musical data

Running JythonMusic Programs

• JEM Editor
• Run, run selection, run line, run paragraph (supports live coding)
• Console (>>>)
• Also, terminal window (low level), and other editors
• developed and maintained by Tobias Kohn

```
x = 4 + 3
x = 7
print "x = ", x
```

```
x = 7
```

• terminal window (Mac shown below; Windows similar)

```
$ cd Dropbox/jythonMusic
$ sh jython.sh furElise.py
```

Outline

• Introduction
• Music Making
• MIDI and OSC
• GUIs
• Image Manipulation
• Timer and Scheduling
• Conclusion

```
# playNote.py
# Demonstrates how to play a single note.
from music import *   # import music library
note = Note(C4, HN)  # create a middle C half note
Play.midi(note)      # and play it!
```
Music Library

• The music library defines functions for giving you control and expression over musical parameters

• They are organized in three categories:
  • Transcription: Note, Phrase, Part, Score
  • Composition: Mod, View, Read / Write
  • Performance (e.g., live coding): AudioSample, MidiSequence, LiveSample, Metronome, Play

Note

• notes contain the simplest possible musical events, consisting of pitch, duration, dynamic, and panning.
  • pitch (0 - 127, also constants, e.g., C4)
  • duration (1.0 = quarter note, also constants, e.g., QN)
  • dynamic (0 - 127, also constants, e.g., FF)
  • panning (0.0 - 1.0)

Note

• how to create notes

\[
\text{\texttt{n = Note(C4, HN)}}
\]

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Note(pitch, duration)</td>
<td>Creates a new note, where pitch is 0-127, and duration is a float (e.g., 1.0 is a quarter note).</td>
</tr>
<tr>
<td>Note(pitch, duration, dynamic)</td>
<td>Creates a new note, where pitch is 0-127, duration is a float, and dynamic is 0-127.</td>
</tr>
<tr>
<td>Note(pitch, duration, dynamic, pan)</td>
<td>Creates a new note, where pitch is 0-127, duration is a float, dynamic is 0-127, and pan is 0.0 (left) to 1.0 (right).</td>
</tr>
</tbody>
</table>

Note

• note functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>n.getPitch()</td>
<td>Retrieves the pitch (0-127) of note n.</td>
</tr>
<tr>
<td>n.setPitch(pitch)</td>
<td></td>
</tr>
<tr>
<td>n.getDuration()</td>
<td></td>
</tr>
<tr>
<td>n.setDuration(duration)</td>
<td></td>
</tr>
<tr>
<td>n.getDynamic()</td>
<td>Retrieves the dynamic (0-127) of note n.</td>
</tr>
<tr>
<td>n.setDynamic(dynamic)</td>
<td></td>
</tr>
<tr>
<td>n.getPan()</td>
<td>Retrieves the pan (0.0-1.0) of note n.</td>
</tr>
<tr>
<td>n.setPan(pan)</td>
<td>Sets the pan (0.0-1.0) of note n.</td>
</tr>
</tbody>
</table>
Pitch

Duration

• note durations are represented using real numbers (floats).

• 4.0 stands for whole note, 2.0 for a half note, 1.0 for a quarter note, etc.

Duration Constants

WHOLE_NOTE = 4.0
DOTTED_HALF_NOTE = 3.0
DOUBLE_DOTTED_HALF_NOTE = 3.5
HALF_NOTE = 2.0
HALF_NOTE_TRIPLET = 4.0/3.0
QUARTER_NOTE = 1.0
QUARTER_NOTE_TRIPLET = 2.0/3.0
DOTTED_QUARTER_NOTE = 1.5
DOUBLE_DOTTED_QUARTER_NOTE = 1.75
EIGHTH_NOTE = 0.5
DOTTED_EIGHTH_NOTE = 0.75
EIGHTH_NOTE_TRIPLET = 1.0/3.0
DOUBLE_DOTTED_EIGHTH_NOTE = 0.875
SIXTEENTH_NOTE = 0.25
DOTTED_SIXTEENTH_NOTE = 0.375
SIXTEENTH_NOTE_TRIPLET = 1.0/6.0
THIRTYSECOND_NOTE = 0.125
THIRTYSECOND_NOTE_TRIPLET = 1.0/12.0

Pitch Constants

C4 | CS4 | D4 | DS4 | EF4 | E4 | ES4 | FS4 | GF4 | G4 | GS4 | AF4 | A4 | AS4 | BF4 | B4 | BS4 | C5
60 | 61  | 62  | 63  | 64  | 65  | 66  | 67  | 68  | 69  | 70  | 71  | 72  |

WN = 4.0, DHN = 3.0, HN = 2.0, DQN = 1.5, QN = 1.0, DEN = 0.75, EN = 0.5, DSN = 0.375, SN = 0.25

WN = 4.0, HN = 2.0, QN = 1.0, EN = 0.5, SN = 0.25
TED Ed - How to read music

Activity #2

```python
# playNote.py
# Demonstrates how to play a single note.

from music import * # import music library

note = Note(C4, HN) # create a middle C half note
Play.midi(note) # and play it!
```

- Play a different note
  - A0 whole (WN) note, half volume (64), center pan (0.5)
  - G5 half (HN) note, full volume (128), extreme right (1.0)
  - rest (REST) whole (WN) note
  - BF3 double whole (WN+WN) note, full volume (128), center (0.5)
    - durations are numbers - can be added, multiplied, etc.

Phrase

- phrases contain a sequence of notes
  - Notes are played sequentially
  - use Rest notes, for gaps

- phrases may also contain chords
  - chords are concurrent pitches with same duration
  - stay tuned...

- phrases have start times
  - float, e.g., 0.0 (in QN time units)
  - no start time means continue after previous phrase (if any)

Phrase

- how to create phrases
### Phrase

- **phrase functions**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>phr.addNote(note)</td>
<td>Appends the given note to the phrase.</td>
</tr>
<tr>
<td>phr.addNote(pitch, duration)</td>
<td>Appends a new note of given pitch (0-127) and duration (a float) to the phrase.</td>
</tr>
<tr>
<td>phr.addNoteList(listOfPitch, listofDurations, listOfDynamics, listOfPanoramics, listofLengths)</td>
<td>Appends the notes specified in terms of pitches (a list), durations (a list), dynamics (a list), penning values (a list), and lengths (a list) to the phrase. The lists are parallel. Dynamic, panoramic, and length lists are optional. Whereas duration determines the printed (notation) value of a note, the length of a note determines how long the note actually sounds.</td>
</tr>
</tbody>
</table>

### Example - Melody

```python
# Example - Melody Theme

# Generate the theme from Beethoven's Fur Elise.
from music import *

# If there are any repetitions, break it up into sections
# Also notice how we set up corresponding pitches and durations

pitch1 = [G, D3, D, G, B, E, C, G]
durations1 = [SN, SN, SN, SN, SN, SN, SN, SN, SN]
pitches2 = [A, REST, C4, E4, A4, B4, REST, E4]
durations2 = [EN, SN, SN, SN, SN, SN, SN, SN]
pitches3 = [G#4, B4, C5, REST, E4]
durations3 = [EN, EN, EN, SN, SN]
pitches4 = [G5, B4, A4]
durations4 = [SN, EN, EN]

# Create an empty phrase, and construct the theme from the above motifs
theme = Phrase()

# Play the generated theme
Play(midi(theme))
```

### Activity #3

- **Write a program that plays the following**

Frère Jacques, Frère Jacques, dor met vous? Dor-mez vous?

Sonnez les ma-ti-nes! Sonnez les ma-ti-nes! Din, dan, don. Din, dan, don. notice the B flat

### MIDI Instruments

- **MIDI has 128 different instruments**

<table>
<thead>
<tr>
<th>Instrument</th>
<th>MIDI Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACOUSTIC_GRAND, PIANO</td>
<td>0</td>
</tr>
<tr>
<td>BRIGHT_ACOUSTIC</td>
<td>1</td>
</tr>
<tr>
<td>ELECTRIC_GRAND</td>
<td>2</td>
</tr>
<tr>
<td>UNTUNED_PIANO, HUMMER_PIANO</td>
<td>3</td>
</tr>
<tr>
<td>EPiano1, Rhodes_PIANO</td>
<td>4</td>
</tr>
<tr>
<td>EPiano2, DX_PIANO</td>
<td>5</td>
</tr>
<tr>
<td>HARP</td>
<td>6</td>
</tr>
<tr>
<td>CLAVINET</td>
<td>7</td>
</tr>
<tr>
<td>CELESTA</td>
<td>8</td>
</tr>
<tr>
<td>GLOCKENPIEL</td>
<td>9</td>
</tr>
<tr>
<td>MUSIC_BOX</td>
<td>10</td>
</tr>
<tr>
<td>VIBRAPHONE, VIBES</td>
<td>11</td>
</tr>
<tr>
<td>MARIMBA</td>
<td>12</td>
</tr>
<tr>
<td>XYLOPHONE</td>
<td>13</td>
</tr>
<tr>
<td>TUBULAR_BELLS</td>
<td>14</td>
</tr>
<tr>
<td>DESKBIEN</td>
<td>15</td>
</tr>
<tr>
<td>DRUMMER_Organ, ORGAN</td>
<td>16</td>
</tr>
<tr>
<td>PERCUSSION_Organ, JAZZ_Organ</td>
<td>17</td>
</tr>
<tr>
<td>ROCK_Organ</td>
<td>18</td>
</tr>
<tr>
<td>CHURCH_Organ</td>
<td>19</td>
</tr>
<tr>
<td>REED_Organ</td>
<td>20</td>
</tr>
<tr>
<td>ACCORDION</td>
<td>21</td>
</tr>
<tr>
<td>HARMONICA</td>
<td>22</td>
</tr>
<tr>
<td>TANGI, ACCORDION, BANDONON</td>
<td>23</td>
</tr>
<tr>
<td>NYLON_GUITAR, GUITAR</td>
<td>24</td>
</tr>
<tr>
<td>STEEL_GUITAR</td>
<td>25</td>
</tr>
<tr>
<td>JAZZ_GUITAR</td>
<td>26</td>
</tr>
<tr>
<td>F/X_O.ooh, GOLIATHS</td>
<td>101</td>
</tr>
<tr>
<td>FX_7_SHARES, ECHO_DROPS</td>
<td>102</td>
</tr>
<tr>
<td>FX_8_Voice, FX_8vertisement</td>
<td>103</td>
</tr>
<tr>
<td>STAGE</td>
<td>104</td>
</tr>
<tr>
<td>BANDO</td>
<td>105</td>
</tr>
<tr>
<td>BANJO</td>
<td>106</td>
</tr>
<tr>
<td>BASSLIKE</td>
<td>107</td>
</tr>
<tr>
<td>FIDDLE</td>
<td>108</td>
</tr>
<tr>
<td>SHANNOIS</td>
<td>109</td>
</tr>
<tr>
<td>TINSEL_BELL, BELL</td>
<td>110</td>
</tr>
<tr>
<td>AGEDO</td>
<td>111</td>
</tr>
<tr>
<td>STEEL_DRUMS</td>
<td>112</td>
</tr>
<tr>
<td>WOODBLOCK</td>
<td>113</td>
</tr>
<tr>
<td>TAPE_DREER, TAPE</td>
<td>114</td>
</tr>
<tr>
<td>HELIO_DOWN, TIM_TOM</td>
<td>115</td>
</tr>
<tr>
<td>SYNTH_Drum</td>
<td>116</td>
</tr>
<tr>
<td>REVIVE_KYMBAUL</td>
<td>117</td>
</tr>
<tr>
<td>GUITAR, PRESET, PRESET_NOISE</td>
<td>118</td>
</tr>
<tr>
<td>PRESET_NOISE, BREATHTONE</td>
<td>119</td>
</tr>
<tr>
<td>ELECTRO_PIANO, SIA</td>
<td>120</td>
</tr>
<tr>
<td>BIRD, BIRD, BIRD</td>
<td>121</td>
</tr>
<tr>
<td>TELEPHONE_EING, TELEPHONE</td>
<td>122</td>
</tr>
<tr>
<td>HELICOPTER</td>
<td>123</td>
</tr>
<tr>
<td>APPLAUSE</td>
<td>124</td>
</tr>
<tr>
<td>GASPERS</td>
<td>125</td>
</tr>
</tbody>
</table>
Activity #4

- Change the default instrument. Try a few different options. Any preferences?

Part

- parts contain phrases to be played by a single instrument
  - phrases played in parallel, may overlap
  - part has a MIDI channel (0-15)
  - part may have a title (string)

- each part assigns MIDI instrument (0-127) to MIDI channel (0-15)
  - one instrument per channel!

- MIDI channel 9 reserved for percussion
  - stay tuned...
Part

• how to create parts

```python
part = Part("An example flute part", FLUTE, 0)
```

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Part()</td>
<td>Creates an empty part.</td>
</tr>
<tr>
<td>Part(instrument)</td>
<td>Creates an empty part with the timbre of the specified instrument</td>
</tr>
<tr>
<td>Part(instrument,</td>
<td>(0 — 127).</td>
</tr>
<tr>
<td>channel)</td>
<td>Creates an empty part with the timbre of the specified instrument</td>
</tr>
<tr>
<td>Part(title,</td>
<td>(0 — 127), and using the specified MIDI channel (0 — 15).</td>
</tr>
<tr>
<td>instrument,</td>
<td>Creates an empty part with the specified title (a string), with the</td>
</tr>
<tr>
<td>channel)</td>
<td>timbre of the specified instrument (0 — 127), and using the specified</td>
</tr>
<tr>
<td></td>
<td>MIDI channel (0 — 15).</td>
</tr>
</tbody>
</table>

Example - String Quartet

```python
# Demonstrate how to create concurrent musical parts.
# No, Opus 64 no 5

from music import *

stringsPart = Part(STRINGS, 0)  # create empty strings part
stringsPart.setTempo(104)

pitches1 = [AS, REST, AS, REST, AS, REST, AS, A6, E6, D6, C6, D6, E6]
pitches2 = [EN, EN, EN, EN, EN, EN, EN, EN, EN, EN, EN, EN, EN, EN]
durations1 = [QN, QN, QN, QN, QN, QN, QN, QN, QN, QN, QN, QN, QN, QN]
durations2 = [QN, QN, QN, QN, QN, QN, QN, QN, QN, QN, QN, QN, QN, QN]

violin1 = Phrase(0.8)  # create a phrase
violin1.addNoteList(pitches1, durations1)  # add notes to the phrase
stringsPart.addPhrase(violin1)

violin2 = Phrase(0.8)  # create a phrase
violin2.addNoteList(pitches2, durations2)  # add notes to the phrase
stringsPart.addPhrase(violin2)  # now, add phrase to part
```

Part

• part functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>part.addPhrase(phrase)</td>
<td>Add a phrase to this part. If the phrase does not have a specific start time, it is added to the end of the part.</td>
</tr>
<tr>
<td>part.addPhraseList(listOfPhrases)</td>
<td>Adds the specified phrases (a list) to the part. If a phrase does not have a specific start time, it is added to the end of the part.</td>
</tr>
</tbody>
</table>

Live Coding with String Quartet
Drums and Percussion

- MIDI channel 9 reserved for percussion
- Pitch of notes (in phrases) in a part going to channel 9, determines which percussive instrument to play
- constants available

Example - Autumn Leaves

```python
# autumnleaves.py
# It plays the theme from "Autumn Leaves", in a Jazz trio arrangement
# using trumpet, vibraphone, and acoustic bass instruments.
# from music import *
# define the data structure (score, parts, and phrases)
from music import *

# Score("Autumn Leaves (Jazz Trio)", 148) # 148 bpm
trumpetPart = Part(TROMPET, 0) # trumpet to MIDI channel 0
vibesPort = Part(VIBES, 1) # vibraphone to MIDI channel 1
bassPort = Part(ACOUSTIC_BASS, 2) # bass to MIDI channel 2

melodyPhrase = Phrase() # holds the melody
chorusPhrase = Phrase() # holds the chorus
bassPhrase = Phrase() # holds the bass line

create musical data
melodyPitch1 = [REST, E4, F5, G5, C5, REST, D4, E4, F5, B4, B4]
melodyPitch2 = [REST, C4, D4, E4, A4, REST, B3, A4, G4, E4]

melodyPhrase.addNote(melodyPitch1, melodyPitch1)
melodyPhrase.addNote(melodyPitch2, melodyPitch2)

# chords
```

Example - Drums

```python
# drumExample.py
# A quick demonstration of playing a drum sound.
# from music import *
# Use constants defined in "drumsconstants.py"

# for drums always use a part on channel 9
# when using channel 9, the instrument (2nd argument) is ignored
drumPart = Part("Drums", 9)

# try one of these pitch: ACOUSTIC_BASS_DRUM, BASS_DRUM1, SIDE_STICK
# ACOUSTIC_SNARE, HAND_CLAP, ELECTRIC_SNARE, LOW_FLOOR_TOM, CLOSED_HI_HAT
# HIGH_FLOOR_TOM, PEDAL_HI_HAT, LOW_TOM, OPEN_HI_HAT, LOW_MID_TOM,
# HI_MID_TOM, CRASH_CYMBAL_1 (for more, see Appendix A)

note = Note( PEDAL_HI_HAT, 1, 9 )
drumPhrase = Phrase(note)
drumPhrase.addNote(note)

print(drumpart)

drumPhrase.addPhrase(drumPhrase)

Play.midi( drumPart )
```
Example - Drum Machine

```python
# drumMachinePattern1.py
# Implements a drum-machine pattern consisting of bass (kick), snare and hi-hat sounds. It uses notes, since phrases, a part and a score, with each layer adding additional rhythms.

from music import *

repetitions = 4  # times to repeat drum pattern

# define the data structure
score = Score("Drum Machine Pattern #1", 125.0)  # tempo is 125 bpm
drumPart = Part("Drums", 0, 0)  # using MIDI channel 9 (percussion)
bassDrumPhrase = Phrase(0.0)  # create phrase for each drum sound
hiHatPhrase = Phrase(0.0)

# create musical data

# bass drum pattern (one bass + one rest 1/4 note) x 4 = 2 measures
bassPitches = [B, B, B, B]  # 4
bassDurations = [Q, Q, Q, Q]  # 4

# snare drum pattern (one rest + one snare 1/4 note) x 4 = 2 measures
snarePitches = [C, C, C, C]  # 4
snareDurations = [Q, Q, Q, Q]  # 4

# create musical data

# bass drum pattern (one bass + one rest 1/4 note) x 4 = 2 measures
bassDrumPhrase.addMelodieList((bassPitches, bassDurations))

# snare drum pattern (one rest + one snare 1/4 note) x 4 = 2 measures
snareDrumPhrase.addMelodieList((snarePitches, snareDurations))
```

Example - Smoke on the Water

```python
# DeepPurple.SmokeOnTheWater.py
# Demonstrates how to combine melodic lines, chords, and percussion. This is based on the intro of "Smoke on the Water" by Deep Purple.

from music import *

# define the data structure
score = Score("Deep Purple, Smoke On The Water", 110)  # 110 bpm
guitarPart = Part("OVERDRIVE_GUITAR", 8)
bassPart = Part("ELECTRIC_BASS", 12)
drumPart = Part(8, 5)  # using MIDI channel 9 (percussion)

# create musical data

guitarPhrases = Phrase(0.0)  # guitar opening melody
guitarPhrase2 = Phrase(32.0)  # guitar opening melody on octave lower
bassPhrase = Phrase(0.0)  # bass melody
drumPhrase = Phrase(0.0)  # drum pattern

# create musical data


guitarDurations = [QN, QN, QN, QN, QN, QN, QN, QN]  # QN=8

guitarPhrase1.addMelodieList((guitarPitches, guitarDurations))
```

Score

- scores contain parts

```python
score = Score("Morning glory", 135.0)
```

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Score()</td>
<td>Creates an empty score.</td>
</tr>
<tr>
<td>Score(title)</td>
<td>Creates an empty score with the specified title (a string).</td>
</tr>
<tr>
<td>Score(tempo)</td>
<td>Creates an empty score with the specified tempo (in beats-per-minute, e.g., 120.0).</td>
</tr>
<tr>
<td>Score(title, tempo)</td>
<td>Creates an empty score with the specified title (a string), and with the specified tempo (in beats-per-minute, e.g., 120.0).</td>
</tr>
</tbody>
</table>

Mod functions

- Mod class provides functions for modifying phrases, parts and scores. These functions modify the data given to them.

  - for example, Mod.repeat() creates repetitions of the given musical material:
    - Mod.repeat(phrase, 41)

- Mod functions are organized by pitch, duration, dynamic, and randomness.
Mod functions

Mod.repeat() - repeats material
Mod.shift() - changes start time
Mod.retrograde() - reverses notes
Mod.transpose() - changes pitch by number of steps
Mod.elongate() - changes note duration by factor
Mod.invert() - mirrors notes along horizontal axis (a pitch)
Mod.palindrome() - appends the retrograde of the notes
Mod.quantize() - rounds start time and duration of notes
Mod.randomize() - randomly changes note aspects
Mod.fadeIn(), Mod.fadeOut() - as name suggests
Mod.normalize() - maximizes volume (maintain differences)
Mod.compress() - maximizes volume (reduce differences)
...etc.

Create a Canon - Row your Boat

```
# rowYourBoat.py
# Demonstrates how to build a musical canon.
# from music import *
# Create the necessary musical data
rowYourBoatScore = Score("Row Your Boat", 108.0) # tempo is 108 bpm
flutePart = Part(FLUTE, 0) # flute part on channel 0
trumpetPart = Part(TRUMPET, 1) # trumpet part on channel 1
clarinetPart = Part(CLARINET, 2) # clarinet part on channel 2
themePhrase = Phrase(8.0) # theme starts at the beginning
# "Row, row, row your boat gently down the stream -
pitches1 = [C4, C4, C4, D4, E4, F4, G4, A4, B4, C5]
durations1 = [QN, QN, QN, QN, QN, QN, QN, QN, QN, QN]
# "merrily, merrily, merrily, merrily"
pitches2 = [C5, C5, C5, G4, G4, G4, C4, C4, C4, C4]
durations2 = [ENT, ENT, ENT, ENT, ENT, ENT, ENT, ENT, ENT, ENT]
# "life is but a dream."
pitches3 = [E4, F4, E4, C4, D4]
durations3 = [DEN, SN, DEN, SN, MN]
```

Activity #5

- Modify your earlier program to create a canon
  - use Mod functions, as per “Row your Boat”
  - use different instruments
  - asterisks (*) below indicate start of new voice
    (total of 4 voices)

Steve Reich - Piano Phase

```
# pianoPhase.py
# Plays Steve Reich's minimal music piece, Piano Phase (1967).
from music import *
# create piano part
pianoPart = Part(PIANO, 0)
# create two phrases
phrase1 = Phrase(8.0)
phrase2 = Phrase(8.0)
# add music in a convenient way
pitchList = [E4, F4, B4, C5, DS, FS, E4, FS, B4, FS, D, C5]
durationList = [SN, SN, SN, SN, SN, SN, SN, SN, SN, SN, SN, SN]
# add the same notes to both phrases
phrase1.addNoteList(pitchList, durationList)
phrase2.addNoteList(pitchList, durationList)
# repeat first phrase 41 times
Mod.repeat(phrase1, 41)
# repeat second phrase 41 times
Mod.repeat(phrase2, 41)
# set tempo to 100 beats per minute
phrase1.setTempo(100.0)
phrase2.setTempo(100.0)
# add phrases to part
pianoPart.addPhrase(phrase1)
pianoPart.addPhrase(phrase2)
# play music, and save it to a MIDI file
Play.midi(pianoPart)
Write.midi(pianoPart, "pianoPhase.mid")
```
The MIDI library defines functions for communicating with MIDI instruments and devices.

These devices need to be physically connected to your computer.

There are two classes: MidiIn, and MidiOut.

To use these classes, you need:

from midi import *

MidiIn objects receive incoming MIDI messages from MIDI devices (e.g., MIDI keyboards, drum pads, etc.)

These devices need to be physically connected (via wire) to your computer.

To create a MidiIn object:

```python
midiIn = MidiIn()
```

creates a MidiIn object to receive incoming MIDI messages from a device.

when called, it presents the user with a GUI to select a MIDI device.
onNoteOn() connects NOTE_ON events with a callback function
- that function should accept four parameters:
  - event type (an integer)
  - channel (0 - 15)
  - data1 (pitch, 0 - 127)
  - data2 (volume, 0 - 127)
- also onNoteOff, onSetInstrument(), onInput()

Example MidiIn Program #2

```python
# midi2.py
from midi import *
midiIn = MidiIn()
def printEvent(eventType, channel, data1, data2):
    print "MIDI message:", eventType, channel, data1, data2
midiIn.onInput(ALL_EVENTS, printEvent)
```

onInput() connects a particular event type (an integer) with a callback function
- that function should accept four parameters:
  - event type (an integer)
  - channel (0 - 15)
  - data1 (an integer, depends on event type)
  - data2 (an integer, depends on event type)
- also note showMessages() and hideMessages()

MidiOut Class

- MidiOut objects send MIDI messages to devices (e.g., synthesizer) that listen for incoming MIDI events
  - these devices need to be physically connected (via wire) to your computer
- To create a MidiOut object:
  midiOut = MidiOut()
  - creates a MidiOut object to send MIDI messages to a device
  - when called, it presents the user with a GUI to select a MIDI device
The OSC (Open Sound Control) library defines functions for communicating with OSC enabled devices (e.g., smartphones, tablets, other computers, etc.).

- These devices may be anywhere on the Internet (even overseas)
- There are two classes: OscIn, and OscOut
- To use these classes, you need:

```python
from osc import *
```
OscIn Class

- OscIn objects receive incoming OSC messages from OSC devices (e.g., an OSC-enabled smartphone, or tablet)

- To create an OscIn object:

```python
oscIn = OscIn( port )
```

- creates an OscIn object to receive incoming messages on port (1024 to 65535, not being used already)

- also make sure firewall permits this port (UPD)

Example OscIn Program #1

```python
# oscIn1.py
from osc import *
oscIn = OscIn( 57110 )
def simple(message):
    print "Hello world!"
oscIn.onInput("/helloWorld", simple)
```

Example OscIn Program #2

```python
# oscIn2.py
from osc import *
oscIn = OscIn( 57110 )
def printMessage(message):
    address = message.getAddress()
    args = message.getArguments()
    print "OSC message!", address,
    for i in range( len(args) ):
        print str(args[i]),
    print
oscIn.onInput("/.*", printMessage)
```

- When you run this program, it will output the following:

  OSC Server started:
  Accepting OSC input on IP address xxx.xxx.xxx.xxx at port 57110  
  (use this info to configure OSC clients)

  - where “xxx.xxx.xxx.xxx” is the IP address of the receiving computer (e.g., “192.168.1.223”)

  - very helpful!
• Notice the special OSC address "/.*"
  • matches all incoming addresses
• onInput() uses regular expressions to specify OSC addresses
• usually simple OSC addresses suffice, e.g.,
  • "/gyro"
  • "/accelerometer", etc.
• also note showMessages() and hideMessages()
Outline

- Introduction
- Music Making
- MIDI and OSC
- GUIs
- Image Manipulation
- Timer and Scheduling
- Conclusion

GUI Library

- The GUI library defines functions for building graphical user interfaces. They are organized as follows:
  - Display
  - Widgets: Label, Button, Checkbox, Slider, DropDownList, TextField, TextArea, Icon, Menu
  - Graphics: Line, Circle, Point, Oval, Rectangle, Polygon, Arc
  - Events: Keyboard Events, Mouse Events, Display Events
  - Color

Display

- to build a GUI, you need at least one display
- displays are application windows
  - contain other GUI objects (widgets and graphics)
  - a program may have several displays open
  - origin (0, 0) is at the top-left corner

```javascript
d = Display("Simple GUI", 120, 60)
```

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Display(title, width, height)</td>
<td>Creates a display window with the specified title (string – default is blank), width (default is 600 pixels), and height (default is 400 pixels).</td>
</tr>
<tr>
<td>Display(title, width, height, x, y, color)</td>
<td>Same as above, but also initial x and y position on screen (default is (0, 0) at top-left), and background color (default is Color.WHITE).</td>
</tr>
</tbody>
</table>
Display

• display functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>d.close()</td>
<td>Closes display d.</td>
</tr>
<tr>
<td>d.show()</td>
<td>Shows display d. This happens automatically, when a new display is created.</td>
</tr>
<tr>
<td>d.hide()</td>
<td>Hides display d.</td>
</tr>
<tr>
<td>d.add(object, x, y)</td>
<td>Adds a GUI object on display d, at coordinates (x, y). It aligns the object's top-left corner (for Circle, its center) with these coordinates. The origin (0, 0) is at the display's top left corner.</td>
</tr>
</tbody>
</table>

Example - Draw Random Circles

```python
# randomCircles.py
# Demonstrates how to draw random circles on a GUI display.
#
from gui import *
from random import *

numberOfCircles = 1000   # how many circles to draw

d = display('Random Circles', 600, 400)

draw various filled circles with random position, radius, color
for i in range(numberOfCircles):
    # create a random circle, and place it on the display
    x = randint(0, d.getWidth()-1)   # x may be anywhere on display
    y = randint(0, d.getHeight()-1)  # y may be anywhere on display
    radius = randint(1, 40)         # random radius (1-40 pixels)

    # get random color (RGB)
    red = randint(0, 255)           # random & (0-255)
    green = randint(0, 255)         # random & (0-255)
    blue = randint(0, 255)          # random & (0-255)
    color = Color(red, green, blue) # build color from random RGB

    # create a filled circle from random values
    c = Circle(x, y, radius, color, True)
```

Graphics

• Graphics objects are used to draw various geometric shapes on a display
  • Line
  • Circle
  • Point
  • Oval
  • Rectangle
  • Polygon
  etc.

Widgets

• Widgets are used to display information and receive user input
  • Label
  • Button
  • Slider
  • Checkbox
  • TextField
  • Icon
  • Menu
  etc.
AudioSample

- AudioSample is used for playing audio files (samples or loops) in real-time
  - may be played once, looped, paused, resumed, and change pitch/frequency, volume, and panning
  - supported audio file formats are WAV and AIF (16, 24 and 32 bit PCM, and 32-bit float)
AudioSample

- how to create AudioSamples

```python
a = AudioSample("sound.wav")
```

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AudioSample(filename, pitch, volume)</td>
<td>Creates an audio sample from the audio file specified in filename (supported formats are WAV and AIF — 16, 24 and 32 bit PCM, and 32-bit float). Parameter pitch (optional) specifies a base MIDI pitch to be associated with the sound (default is A4), so we can play different pitches with it by changing its pitch (or frequency). Parameter volume (optional) specifies a MIDI note velocity to be used for playback (default is 127).</td>
</tr>
</tbody>
</table>

AudioSample functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>a.play()</td>
<td>Play the sample once. If start and size are provided, the sample is played from millisecond start until millisecond start-size (default is 0 and -1, respectively, meaning from beginning to end).</td>
</tr>
<tr>
<td>a.play(start, size)</td>
<td></td>
</tr>
<tr>
<td>a.loop()</td>
<td>Repeat the sample indefinitely. Optional parameters times specifies the number of times to repeat (default is -1, indefinitely). If start and size are provided, looping occurs between millisecond start and millisecond start-size (default is 0 and -1, respectively, meaning from beginning to end).</td>
</tr>
<tr>
<td>a.loop(times, start, size)</td>
<td></td>
</tr>
<tr>
<td>a.stop()</td>
<td>Stops sample playback immediately.</td>
</tr>
<tr>
<td>a.pause()</td>
<td>Pauses sample playback (remembers current position for resume).</td>
</tr>
<tr>
<td>a.resume()</td>
<td>Resumes sample playback (from the paused position).</td>
</tr>
<tr>
<td>a.isPlaying()</td>
<td>Returns True if the sample is still playing, False otherwise.</td>
</tr>
<tr>
<td>a.setPitch(pitch)</td>
<td>Sets the sample pitch (0-127) through pitch shifting from sample’s base pitch.</td>
</tr>
</tbody>
</table>

Activity #6

- Download a few audio files, and modify the last program to loop them, and change their frequency and volume. Perform your “composition”.

- Get sounds from
  - https://www.freesound.org/
  - http://soundbible.com/
  - many other sites

Example - Circle Instrument
Example - Virtual Piano

Outline

- Introduction
- Music Making
- MIDI and OSC
- GUls
- Image Manipulation
- Timer and Scheduling
- Conclusion

Image Library

- The image library defines functions for displaying and manipulating JPEG and PNG graphic files

Image Class

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Image(filename)</td>
<td>Reads in a .jpg or .png file called filename (a string) and shows an image. It returns the image, so it should be stored in a variable, e.g., &lt;br&gt;img = Image(&quot;sunset.png&quot;)</td>
</tr>
<tr>
<td>Image(width, height)</td>
<td>Returns an empty (blank) image with provided width and height. It returns the image, so it should be stored in a variable, e.g., &lt;br&gt;img = Image(300, 300)</td>
</tr>
<tr>
<td>img.getWidth()</td>
<td>Returns the width of image img.</td>
</tr>
<tr>
<td>img.getHeight()</td>
<td>Returns the height of image img.</td>
</tr>
<tr>
<td>img.getPixel(col, row)</td>
<td>Returns this pixel’s RGB values (a list, e.g., [255, 0, 0]), where col is the image column, and row is the image row. The image origin (0, 0) is at top left.</td>
</tr>
<tr>
<td>img.setPixel(col, row, RGBlist)</td>
<td>Sets this pixel’s RGB values, e.g., [255, 0, 0], where col is the image column, and row is the image row. The image origin (0, 0) is at top left.</td>
</tr>
</tbody>
</table>
Outline

- Introduction
- Music Making
- MIDI and OSC
- GUIs
- Image Manipulation
- Timer and Scheduling
- Conclusion

Timer Library

- The timer library defines functions for scheduling tasks (e.g., playing sounds, animation)
Timer Class

• creates a Timer t, which will call function Play.noteOn(A4) repeatedly every 500 milliseconds (i.e., half second)

For example, the following:

\[ t = \text{Timer}(500, \text{Play.noteOn}(A4), \text{True}) \]

• creates a Timer t, which will call function Play.noteOn(A4) repeatedly every 500 milliseconds (i.e., half second)

Outline

• Introduction
• Music Making
• MIDI and OSC
• GUIs
• Image Manipulation
• Timer and Scheduling
• Conclusion

Conclusion

• for more info, see
  http://jythonmusic.org

• also see