Chapter 2
Music Programming Environments

Abstract This chapter introduces the music programming environments that will be used throughout the book. We start by giving an overview of the roles each language might play in the development of computer instruments. Following this, we focus on the three specific languages we will employ, Python, Csound, and Faust. Python can be used in its role as a glue and host language containing dedicated code written in the Csound language. It can also be employed for complete signal processing programs. Csound is a prime DSL for music, which can be used in a variety of ways. Finally, we introduce Faust as a way of programming dedicated signal-processing algorithms that can be placed within the Csound environment or used as stand-alone programs. The chapter concludes with an overview of how these three systems can coexist in an instrument development scenario

Computer music instruments have a long history of software support. It stretches all the way back to the early 1960s and the introduction by Max Mathews of his MUSIC III program for the IBM 7904 computer [63]. This was an unprecedented invention, an *acoustical compiler*, as Mathews called it, which provided the form for a multitude of computer music programming environments that were to follow. His idea, informed by earlier experiments with MUSIC I and II, was that musicians, composers, researchers, etc. would be better served by a system that had open possibilities, based on building blocks that could be connected together. In other words, rather than a program that can generate sound given parameters, one that can create new programs for that purpose [88].

His acoustical compiler was able to create software based on a given description that could be employed in music composition: a computer instrument [45]. Mathews followed it with MUSIC I [64, 100], and MUSIC V [65]. From this lineage of software, also known as the MUSIC N family, there are a number of systems that are currently in use. The one we will be employing in this book is *Csound* [59], which will be introduced in this chapter. In Csound, we have a complete domain-specific language (DSL), and a performance environment to run compiled programs that is unrivalled in terms of its breadth of sound-processing possibilities.

V. Lazzarini, *Computer Music Instruments*,
https://doi.org/10.1007/978-3-319-63504-0_2

Alongside the very domain-specific programming environments offered by the MUSIC N-derived software, we should also consider the role of general-purpose programming languages (GPPLs) in the development of computer instruments. It is true to say that we could dispense completely with domain-specific approaches, because any language could be employed for this purpose. In practice, however, we will see that it is far more productive to use a DSL for specific aspects of instrument development (e.g. the signal-processing elements of it). A GPPL might not have certain components built into it that would facilitate their design and implementation.

However, there are two roles into which GPPLs could be slotted quite well. The first is to provide a glue for control and signal-processing components of our system. For example, to allow the user to create a host user interface, graphical or otherwise, to a collection of instruments. In this instance, while, again, many languages could be chosen, the most appropriate ones are the high-level scripting languages. This is because they are very straightforward to use and generally provide many possibilities for interfacing, either built in as part of the programming environment or as loadable modules. In this book, we will be employing the *Python* programming language for this purpose, as it has all the attributes listed above and, in addition to these, has excellent support for integration with Csound. Python, on its own, will serve as a means of demonstrating signal-processing algorithms, as it can provide compact code with excellent readability.

The second role in which we could find the use of a GPPL is at lower level in the implementation of computationally-intensive tasks. In this case, we are better served by a language that is classed as implementation level, such as C or C++. Specific elements of an instrument that are more demanding in terms of resources should ideally be programmed using these languages and compiled into components that can be loaded by the DSL. In this book, we will be using these languages indirectly, by employing a signal-processing language, *Faust* [77], which can be compiled into efficient C++ code, or directly into a loadable object using a just-in-time (JIT) compiler.

This chapter will first consider these three systems, Csound, Python, and Faust, separately, providing an informal overview of their languages and programming. Following this, the ways in which these can be integrated into a single workflow will be discussed. These languages will be then used in examples throughout the book.

2.1 Python

Python is a scripting language employed in a variety of applications, from user interfaces to scientific computing. It is a so-called multi-paradigm language, allowing a diverse variety of programming approaches. It is generally simple to use and allows for quick prototyping of ideas. Since it relies on an *interpreter* to translate programs, it tends to be slow for computationally-intensive applications. However,

it is very easily extendable, which allows it to use pre-built components to speed up some tasks.

The system (language/interpreter) has witnessed a major version change in recent years. It has moved from version 2 (which was kept at 2.7) to 3 (currently 3.5) in a non-backward compatible way. This has confused many users, as it took a long time for many popular third-party components to be upgraded. At the time of writing, Python 3 has become well established with most of the important dependencies being made available for it. In this book, we will be adopting this version. This section will introduce the key language elements that will allow the reader to follow the program examples in later chapters. Further information can be found in the official language reference documentation at python.org.

Python can be used interactively by typing code directly into an interpreter shell, or by supplying it with programs written in text files. For instance the following command starts an interactive session:

```
$ python3
Python 3.5.1 (v3.5.1:37a07cee5969, Dec  5 2015,
21:12:44)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for
more information.
>>>
```

At this stage, commands can be typed at the prompt (>>>). Programs can also be passed as files to the command:

```
$ python3 helloworld.py
Hello World!!
```

In this case, the file `helloworld.py` consists of a single line:

```
print('Hello World!!')
```

Python is a dynamically-typed language. This means that variable types are defined according to the data they are carrying and the type can be re-defined for the same name, if we need to:

```
>>> a = 'hello'
>>> type(a)
<class 'str'>
>>> a = 1
>>> type(a)
<class 'int'>
```

In this example the variable `a` was first holding a text string, and then a whole number, which represent two distinct types. In addition to the various types we will discuss below, Python has a special one, `None`, which indicates the absence of a value and is used in situations when we need to specify this explicitly.

Built-in numeric data in Python is divided into three basic categories: integers (`<class 'int'>`), floating-point numbers[1] (`<class 'float'>`) and complex numbers (`<class 'complex'>`). Integers have unlimited precision and floating-point data is implemented using double precision. Complex numbers have two parts, real and imaginary, both of which are floating-point. A complex constant has the form `a + bj`, where the letter `j` is used to label the imaginary part.

In addition to these, a special type of numeric data are the boolean numbers. These are used in truth expressions and can assume two values: `True` or `False`. These are a subtype of integers and behave like 1 and 0, respectively. When converted to text, they become the strings `'True'` and `'False'`.

Arithmetic statements are very similar to various other languages. Assignment is defined by =, and the common arithmetic operators are as usual: `-`, `+`, `*`, `/`, and `\%` minus/difference, addition, multiplication, division, and modulus, respectively. To these, Python adds `**` for exponentiation and `@` for matrix multiplication. In Python 3, division always returns a floating-point number even if we are using integer operands. To force truncation, we can use a double slash (`//`). As usual, expressions can be combined together with parentheses (`( )`) to determine the precedence of operations. Some of these operations can also have different meanings in contexts that are not arithmetic, as we will see at various times in this book.

One significant aspect of the Python syntax is its insistence on indentation levels as code block delimiters. This typographical approach works well for code readability, but it is awkward in interactive use. For instance, to define a callable subroutine, we use the the keyword `def` followed by the routine name, a list of parameters, and a colon. The code that is going to be executed by this subroutine has to be placed at a higher level of indentation to indicate that it belongs to the subroutine:

```
def  mix(a,b):
    c  = a+b
    return c
```

All the code that belongs together in a block needs to be at the same level of indentation. This is also the case in other syntactic constructions that we will see later in this chapter. Finally, comments starting with a hash (#) and continuing to the end of the line are allowed.

2.1.1 Control of flow and loops

As with other languages, control of flow can be achieved with a combination of `if` – `elif` – `else` statements, which evaluate a truth expression. Code in sections defined by these keywords needs to be formatted at a higher indentation level. For example:

```
if a < 0:
```

[1] The floating-point data type is used to represent real numbers in programs.

```
    print('negative')
elif a == 0:
    print('zero')
else:
    print('positive')
```

As usual, the logical expressions following the `if` and `elif` keywords can have multiple terms, combined by use of the boolean logical operators `and` and `or`.

The basic looping construction in Python is defined by `while`, followed by a truth expression and a colon. The loop block is then placed, indented, under this:

```
n = 0
while n < 10:
  print(n)
  n += 1
```

Here, we can optionally place an `else` section after the loop body, which will execute when the expression is not true.

Another form of loop can be found in the `for ... in ...` statement. This is slightly more complex in that it involves the concept of an *iterable* object that is used to construct the loop. This type of loop is designed to work with sequences such as lists and strings. Here is an example of this construction:

```
for x in [0,1,2,3,4]:
   print(x)
```

which, when executed, yields

```
0
1
2
3
4
```

We can get gist of it from this example. The variable `x` takes the value of a different member of the list each time the body of the loop is executed.

2.1.2 Sequences

In Python there are a number of data types that can be sequentially organised and manipulated, which are known generically as *sequences*. Among these we have lists, text strings, tuples and ranges.

Lists

A list is a sequential collection of data, which can be of the same or different types. Lists are mutable, i.e. they can be enlarged, shortened, have items deleted, replaced,

inserted, etc. To create a list we can use square brackets as delimiters, or use the named constructor:

```
l = [0,1,2,3,4] # list with 5 items
k = []          # empty list
k = list()      # using the named list constructor
```

Lists can be accessed using a similar square-bracket notation. With it, we can take slices of the list, as well as individual items:

```
>>> a = [0,1,2,3,4]
>>> a[0]
0
>>> a[2:4]
[2, 3]
>>> a[0:5:2]
[0, 2, 4]
>>> a[:5:2]
[0, 2, 4]
>>> a[1:]
[1, 2, 3, 4]
>>> a[2] = 100
>>> a
[0, 1, 100, 3, 4]
```

The general form for the slice is `[start:end:increment]`. In addition to this, there are a number of operations that can be used to insert, append, copy, delete, pop, remove, reverse, and sort a list. In addition to these, we can also have *list comprehensions* of the following form:

```
>>> [x*x for x in [1,2,3,4]]
[1, 4, 9, 16]
```

where an expression involving a variable (such as `x*x`) can be iterated over a list input to produce an output list. This is a concise way to create a list based on the evaluation of a function or expression.

Tuples

Tuples are immutable[2] sequence types that can be used to store data objects together. Access is similar to that for lists, and slices are also possible. However, none of the mutation operations are available for tuples. They can be created using a comma-separated sequence of items, or using the named constructor:

```
l = 0,1,2,3,4           # tuple with 5 items
k = tuple([0,1,2,3,4])  # using the named constructor
```

[2] This means that they cannot be changed.

Ranges

The range type is also immutable and represents a sequence of integers with a start and an end. Ranges can be constructed with the forms `range(end)` or `range(start,end,increment)`, where the end value is not actually included in the sequence. To make a list out of a range it is necessary to create one using a list *constructor* (`list()`):

```
>>> list(range(5))
[0, 1, 2, 3, 4]
>>> list(range(0,5,2))
[0, 2, 4]
>>> list(range(1,6))
[1, 2, 3, 4, 5]
```

Text strings

One special case of a sequence is a string. This is another immutable type, which is used to hold text characters. Strings can be single quoted, embedding other double quoted strings; double quoted, embedding other single-quoted strings; or triple quoted, containing multiple lines:

```
>>> a = 'this is a "string"'
>>> a
'this is a "string"'
>>> a = "'this' is a string"
>>> a
"'this' is a string"
>>> a = '''these are lines
... lines and
... lines'''
>>> a
'these are lines\nlines and\nlines'
>>> print(a)
these are lines
lines and
lines
```

A number of operations can be performed on strings. For instance, the addition operator (+) can be used to concatenate strings:

```
>>> a = 'hello'
>>> b = ' world!!'
>>> a+b
'hello world!!'
```

Strings can be formatted in ways similar to the standard C library function `printf` using the `%` operator. For instance:

```
>>> a = "this is a float: %f and this is an integer: %d"
>>> a % (1.1, 2)
'this is a float: 1.100000 and this is an integer: 2'
```

2.1.3 Functions

Functions, an example of which has already been presented earlier in this chapter, can be used to implement subroutines or proper functions, with or without side-effects. The basic syntax is:

```
def  function-name(argument-list):
     function-body
```

The function body has to be indented at a higher level to make it a separate block. Alternatively, if the function consists of one line only, then it can be place on the same line after the colon. The comma-separated argument list can have initialisers, or default values, for parameters, and the function body can include the keyword `return` to return results:

```
def  add_one(a=0):
     return a+1
```

Functions can return multiple values as tuples:

```
def  add_one_two(a=0):
     return a+1,a+2
```

On a function call, arguments are passed by *object reference*. This means that a name on the argument list refers to a data object[3] that is passed at the function call and it can then be used in the function body accordingly. For instance:

```
def  func(a):
     print(a)
```

The name `a` refers to whatever object was given to the function and then is passed to `print`. This might be of different types and the function will adjust its behaviour accordingly:

```
>>> func('hello')
hello
>>> func(1)
1
```

[3] All data in Python are objects of one type or another: numbers, sequences, text strings, etc. are examples of built-in types. Other types can be user-defined, or supplied by packages.

Arguments can also be passed out of order with keywords, which default to the argument names:

```
>>> def func(a,b): print(a+b)
...
>>> func(b='world!!', a='hello ')
hello world!!
>>>
```

The scope of the names defined in a function is local, or within the function body or block. On assignment, an object is created and bound to a name, locally. If we want to keep it, we have to return a reference to it:

```
def  func(a):
     b = a + 1 # an object is created and assigned to b
     return b  # the object reference is returned
```

However, it is possible to access objects in the global namespace directly from within a function, but only if the given name is not used in the function:

```
>>> def func(): return b + 1
...
>>> b = 2
>>> func()
3
>>> b
2
```

We can also use a special keyword `global` to refer explicitly to a global name. In any case, we should avoid directly accessing objects that are in the global scope.

2.1.4 Classes

Python supports the creation of new types of objects with dedicated operations. This is done through class definitions:

```
class NewType:
  ...
```

Classes are, by default, derived from a basic type called `object`, unless we explicitly pass another type in the declaration:

```
class NewType(ParentType):
  ...
```

Following the class declaration, we normally place a number of methods to operate on this new type. These are special types of functions that are bound to the class and can manipulate it:

```
class NewType:
    def method(self):
            ...
```

The first argument to a class method is always the class object (by convention, we call it `self`) and can be used to refer to all the class members (variables and functions, or *methods*). In addition to this argument, we can have a normal argument list. Accessing methods of a given instance of a class use the syntax `object.method()`, where `object` is the variable name of a given class, and `method` one of its member functions. Classes can also have data members, or instance variables, and these can be created by methods, added to the class definition or even added to a new object after its creation. These can also be accessed using a similar syntax, `object.variable`.

An object of a newly created class can be created by invoking its constructor, which is a function with the name of the class:

```
a = NewType()
```

Constructors can be declared as a method named `__init()__`. This is used to perform custom initialisation tasks:

```
class NewType:
    def __init__(self,a):
         self.a = a
    def print(self):
         print(self.a)
```

In this case, the constructor allows us to create objects with different attributes:

```
>>> a = NewType(1)
>>> a.print()
1
>>> a = NewType(2)
>>> a.print()
2
```

Inheritance allows a derived class to access methods and variables in the parent class as if they were its own. A specific method in the base class can be accessed using the syntax `Parent.method()`, if necessary.

2.1.5 Modules and libraries

Python can be extended with a variety of specialised modules or libraries. Normally these are available from the central repositories at python.org or through package

management systems[4]. Once a module/package is installed, we can load it using the `import` statement:

```
import package_name
```

From there, all code in a package `pkg` can be accessed using the form `pkg.obj`, where `obj` stands for any method, variable, constant, etc. defined in the package. We can also give an alias `name` to a package:

```
import package_name as name
```

Pylab

In this book, we will be using a number of external libraries. In particular, we will employ the scientific libraries SciPy, NumPy, and matplotlib, packaged together as pylab. This will be commonly invoked by the command:

```
import pylab as pl
```

Most of the key SciPy/NumPy/matplolib objects will be exposed through this interface. Occasionally, we might need to load specific parts of these libraries separately, if they are not available through pylab. The fundamental data type that is used by all of these libraries is the NumPy `array`, which has been designed to optimise numeric computation. It behaves very similarly to other sequence types (such as lists), but it actually holds a numeric array in native form (integers, float and complex subtypes are provided).

2.2 Csound

Csound is a sound and music computing system that can be used in a variety of applications. It is programmed and run via its DSL through a variety of frontend software. In this section we will introduce the basic aspects of its operation and the main features of its language. We will concentrate on the most relevant elements that will allow the reader to follow the examples in the next chapters. Further concepts will be introduced as required. A complete reference for the system and language can be found in [59].

The Csound language is compiled into an efficient set of signal processing-operations that are then performed by its audio engine. All operations to compile and run code can be accessed through a high-level programming language interface, the Csound Application Programming Interface (API). We will first look at the most salient aspects of the language before discussing how to use the API (through Python) to run the system.

[4] Many operating systems have their own package managers for adding new components. In addition, Python has pip (https://pip.pypa.io/en/stable/), which is used specifically for Python packages on multiple platforms.

2.2.1 The language: instruments

The basic structuring unit in Csound is the *instrument*. This is used to hold a series of operators called *unit generators* that are defined by various *opcodes* in the system. An instrument is a model of a sound-processing/generating object that can be instantiated any number of times. An instrument is defined as follows:

```
instr ID
...
endin
```

where `ID` is an integer or a name identifying the instrument. Inside instruments we place unit generators that can process audio. For example, the opcode `oscili` generates a sine wave signal with a specified amplitude and frequency (the first and second parameters). This opcode takes frequencies in cps (Hz) and amplitudes within the range specified by Csound. This is set by the constant `0dbfs`, which determines the maximum full-scale peak amplitude in the system (see section 1.2.1). It defaults to 32,768 (for backwards compatibility), but it can be set to any positive value. To get audio out of this instrument, we use the opcode `out`:

```
instr 1
  out(oscili(0dbfs/2,440))
endin
```

To run this instrument, we need to schedule it, passing the instrument name, the start time and the duration to `schedule`

```
schedule(1,0,1)
```

The call to `schedule` is placed outside instruments in global space. This call is executed once Csound starts to perform. It could also be placed inside an instrument, in which case it would recursively schedule itself.

2.2.2 Variables and types

Csound has a number of variable types. These define not only the data content, but also the action time. When an instrument is run, there are two distinct action times: initialisation and performance. The former is a once-off run through the code that is used to initialise variables and unit generators, and execute code that is designed to run once per instance. Performance time happens when instruments process signals, looping over the unit generator code.

In the example above, at init time, the oscillator is allocated and its initialisation routine is executed. At perf time, it produces the audio signal that is sent to the output. Signals can be of two types: scalars and vectors. The first type is used for control operations (e.g. parameter automation) and the second for audio. The two

Table 2.1 Basic variable types in Csound.

variable type	action time	data format
i	init	scalar
k	perf	scalar
a	perf	vector

action times and the two types of signals yield three basic variables types: i, k and a (see Table 2.1).

Types are defined by the first letter of the variable names: i1, kNew, asig, etc. We can rewrite our first example using these:

```
instr 1
   iamp = 16000
   icps = 440
   asig = oscili(iamp, icps)
   out(asig)
endin
```

In this case, the first two lines will execute at i-time, followed by the initialisation of the oscillator. At performance time the two last lines will be run in a loop for the requested duration, filling and consuming audio signal vectors. The size of these vectors is determined by the system constant `ksmps` and defaults to 10. The sampling rate is also set by the constant `sr` and its default value is 44,100. A a third quantity, the control rate (`kr`) is set to $\frac{sr}{ksmps}$, determining how often each control (scalar) variable is updated. This also determines the processing loop period (how often the processing code is repeated).

The scope of variables is local to the instrument. Global variables can be created by attaching a 'g' to the start of the variable name (e.g. gi1, gkNew, or gasig). Instruments can contain external initialisation arguments. These are identified by parameter numbers $p1 \ldots pN$. The first three are set by default to the instrument number, start time and duration (in seconds, with -1 indicating indefinite duration). The following arguments can be freely used by instruments. For instance:

```
instr 1
   iamp = p4
   icps = p5
   asig = oscili(iamp, icps)
   out(asig)
endin
```

This will allow us to use parameters 4 and 5 to initialise the amplitude and frequency. In that case, we can have sounds of different pitches:

```
schedule(1,0,1,0dbfs/3,440)
schedule(1,0,1,0dbfs/3,550)
schedule(1,0,1,0dbfs/3,660)
```

Control variables are used to make parameters change over time. For instance, we could use a trapezoidal envelope generator (`linen`) to shape the amplitude of our example instrument, cutting out the clicks at the beginning and end of sounds:

```
instr 1
   idur = p3
   iamp = p4
   icps = p5
   ksig = linen(iamp,0.1,idur,0.2)
   asig = oscili(ksig, icps)
   out(asig)
endin
```

With these three basic types, it is possible to start doing signal processing with Csound. More advanced types will be introduced later when we need them to design certain instruments.

There are two alternative syntaxes in Csound. The examples above demonstrate one of them, where opcode parameters are passed as comma-delimited lists inside parentheses and the output is put into variables using the assignment operator (=). The other form uses an output–opcode–input format, with no parentheses and no explicit assignment, only spaces:

```
instr 1
   idur = p3
   iamp = p4
   icps = p5
   ksig linen   iamp,0.1,idur,0.2
   asig oscili  ksig, icps
        out     asig
endin
```

The two ways of writing instrument code can be combined together. In particular, the function-like syntax with the use of parentheses can be used to inline opcodes in a single code line:

```
   asig oscili linen(iamp,0.1,idur,0.2),icps
```

When opcodes are used with more than one output, however, this syntax is not available, and we need to employ the more traditional output-opcode-input format.

Csound code can be run directly from its various frontends. The most basic one is the `csound` command, to which code can be supplied as a text file. For instance, if our instrument and schedule lines above are written to a file called *instruments*, we could run it in this way:

```
$ csound --orc instruments -o dac
```

where the `-o dac` option tells Csound to play the sound directly to the soundcard. Alternatively, we could use the API to compile and run programs directly.

2.2.3 The API

The Csound API provides complete access to the system, from configuration to code compilation and audio engine performance. It is based on a C/C++ set of functions and classes, but it has been wrapped for various languages. In the case of Python, there are two alternative packages that can be used, csnd6 and ctcsound. The former is only available for Python 2 and the latter can be used in both versions 2 and 3.

ctcsound

The ctcsound package provides full access to the Csound C API, in addition to a few extra components. It can be loaded with the command:

```
import ctcsound
```

A typical use of the package involves the following steps:

- Creating a Csound object
- Setting up options to control the compiler and engine
- Compiling code
- Starting and running the audio engine
- Interacting with the compiled instruments

For example, the following code creates a Csound object, sets it to output audio to the soundcard, compiles some code and performs it:

Listing 2.1 A simple Python Csound API example.

```
import ctcsound

code = '''
instr 1
   idur = p3
   iamp = p4
   icps = p5
   ksig = linen(iamp,0.1,idur,0.2)
   asig = oscili(ksig, icps)
   out(asig)
endin

schedule(1,0,1,0dbfs/3,440)
schedule(1,1,1,0dbfs/3, 550)
schedule(1,2,1,0dbfs/3, 660)
'''

cs = ctcsound.Csound()
cs.setOption('-odac')
```

```
cs.compileOrc(code)
cs.start()
cs.perform()
```

When running the Csound engine, it is often simpler to start a new thread for performance, so that the user can interact with it. For this purpose, we have the `CsoundPerformanceThread` class, which takes a Csound object and allows it to be played:

Listing 2.2 Using a performance thread

```
cs = ctcsound.Csound()
cs.setOption('-odac')
cs.compileOrc(code)
cs.start()
perf = ctcsound.
  CsoundPerformanceThread(self.cs.csound())
perf.play()
```

Methods for pausing, stopping and joining the thread are available (`pause()`, `stop()`, and `join()`). With Csound running in a separate thread, we can start new instruments running and set controls in them using methods of the Csound class. Both of the above examples (with and without threads) can be used to run the Csound code examples in this book, with Python as the host environment. In this case all we need to do is to replace the triple-quoted string in `code` by the program in question. We might also need to set other options, which we can do by calling `setOption()` more than once with the option as an argument. A full working example demonstrating these possibilities will be shown in section 2.4.

The software bus

Communication with a running Csound object is achieved via the software bus. It works through logical *channels* that can be used to transmit control or audio data to and from the host. The principle is straightforward. A channel with any given name can be declared in Csound with the opcode `chn_k` or `chn_a` for control or audio, respectively:

```
chn_a(Sname, imode)
chn_k(Sname, imode)
```

where `Sname` is a string with the channel name and `imode` is the mode (1, 2, 3 for input, output, and input/output respectively). If a channel is not declared, it is created when it is used for the first time.

From Csound, we can write and read with the following opcodes

```
xs chnget Sname
chnset xs, Sname
```

where `xs` is either an i-, k-, or a-type variable containing the data to be written (`chnset`) or to where we should read the contents of the channel (`chnget`).

The Python host uses the following methods of a Csound object to access a channel:

```
# get a value from a channel
Csound.controlChannel(name)
# set a value of a channel
Csound.setControlChannel(name,val)
# get a signal from an audio channel
Csound.audioChannel(name, samples)
# put a signal in an audio channel
Csound.setAudioChannel(name, samples)
```

The software bus can be used throughout a performance and it is particularly useful for the design of user interfaces for real-time instruments (see section 2.4 for a complete example).

Scheduling events

A host can schedule events in a running object in the same way as we have used a `schedule()` opcode inside Csound code. This can be done via the `inputMessage()` method of either the Csound or the CsoundPerformanceThread classes:

```
Csound.inputMessage(event)
CsoundPerformanceThread.inputMessage(event)
```

The argument `event` is a string containing a message that starts with the letter 'i' (for 'instrument'). Parameters are separated from one another by empty spaces (not commas, since they are not function arguments, but items of a *list*). For instance, a Csound code line `schedule(1,0,-1,0.5,400)` can be translated into the following event string:

```
event = 'i 1 0 -1 0.5 400'
```

In this case, we are telling Csound to instantiate instrument 1 immediately (p2 = 0) for an unlimited duration (p3 = −1, *always on*) passing 0.5 and 400 as parameters 4 and 5, respectively. With both scheduling of events and software bus channels, we cancreate complete graphical user interface instruments with Csound and Python, as demonstrated in Sect. 2.4 and in Appendix B.1.

2.3 Faust

Faust [77] is a system designed to manipulate audio streams, and with which we can implement a variety of digital signal processing algorithms. Programs created

in Faust can be more efficient than equivalent code written in other high-level music programming languages. These are generally used as components for specialised purposes in host environments (such as Csound), although stand-alone applications can also be developed. Programs can also be used to generate graphic flowcharts that can assist with the design and implementation of algorithms.

Faust code describes a process on an audio stream in a purely functional form. The following minimal example demonstrates thd basic principle:

```
process = + ;
```

Faust statements are terminated by a semicolon (`;`). The arithmetic operator (+) takes two inputs, mixes them and produces one output. It is implicit in this operation that it takes two inputs and produces one output. Similarly, an example of a minimal program with one input and one output is given by

```
process = *(2);
```

where multiplication by 2 is used. This is a function that maps one input to one output.

2.3.1 Stream operators

Faust has a number of specialised operators to work on streams. Sequential (`:`) and parallel (`,`) operators can be used to organise the order of function application. For instance:

```
process = _ ,2 : *;
```

takes audio input (using _) and the constant 2 at the same time and applies them in a sequence (`:`) to the function '*'. The (_) is a placeholder for a signal.

Two other important operators are split (`<:`) and merge (`:>`), for sending one signal into parallel streams and for mixing two inputs into one stream, respectively. The following example,

```
process =  _ <: *;
```

squares a signal, and

```
process = _,_:> _;
```

mixes two streams.

The '@N' delay operator applies a delay of N samples to a signal, with which we can create an echo effect:

```
process = _<:_@4410,_:> _;
```

Similarly, the single quote and double quote operators apply one- and two-sample delays, respectively. The following program,

```
process = _<: _ +_' : *(0.5);
```

applies averaging to consecutive samples in an input signal.

Complementing these operations, Faust includes the concept of *feedback*, denoted by the '~' (tilde) operator. This is used to feed the output of a stream back into its input.:

```
process = _ + _ ~ _;
```

To the left of the tilde we have the input(s) to the operator. The first one is always the feedback signal, and so if we want to add an input, we need to provide a second one, which in this case is mixed with the feedback. To the right of the tilde we have a feedback path that implies a 1-sample delay. On this side we can place other operations if needed (e.g. scaling, futher delays, etc.). A simple example is given by

```
process =    @(4409)+_ ~ *(0.9);
```

which will mix an input signal with its feedback delayed by 4409 samples. The feedback signal is attenuated by gain (0.9). The total delay is 4410 samples (4409 + the 1-sample delay from the feedback), which at $f_s = 44100$ (the default sampling rate) equals 0.1 sec. This process implements an echo with feedback (multiple repetitions of the input signal). All of these Faust operations are illustrated with flowcharts demonstrating each process in Fig. 2.1. These were generated directly from the Faust process definitions.

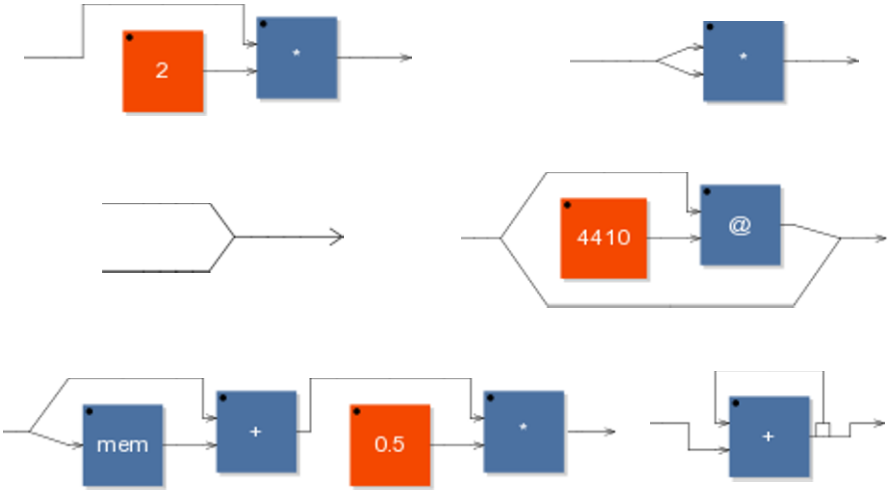

Fig. 2.1 Faust operators. Top row: parallel-sequential (scaling by 2); split (squaring). Mid row: merge (mix); delay (@). Bottom row: one-sample delay (single quote); and feedback (tilde).

2.3.2 Functions

Functions can be defined to conveniently describe an operation. For instance, we could rewrite the averaging process in a clearer form as

```
f(x) = (x + x')/2;
```

which can be used at various places in a program. Anonymous (lambda) functions can also be used using the following notation:

```
\(x).((x + x')/2);
```

This can be used to simplify the notation of some algorithms, when the direct use of stream operations can make the code look cluttered.

2.3.3 Controls

Faust also allows the building of generic user interfaces (UIs). These can take different forms depending on the actual platforms on which the code can be compiled. As an example, a generic horizontal slider is defined by

```
freq = hslider("frequency", 440, 100, 1000, 1);
```

which takes as parameters a name label, a default value, a minimum, a maximum, and a minimum step.

2.3.4 Compilation and Csound integration

Faust programs can be compiled into C++ code for a variety of *architectures*, including Csound plugin opcodes. From the generated source code, we can build the library binaries that can be loaded into the system. The Faust program can then be used in Csound instruments as a new opcode.

Alternatively, Csound includes a set of opcodes designed to take Faust code directly to compile and run it [48]. In this case, it is possible to integrate the two systems intp a very tightly-knit arrangement. There are three basic steps in this process:

1. Compilation: Faust code is compiled into a DSP factory object. This is a binary representation of the processing code that can be instantiated.
2. Instantiation: from a DSP factory object, we can create a process instance, which is ready to be run and to perform the desired signal processing.
3. Performance: with an instance in memory, it is possible to invoke its `compute()` to produce the audio output.

The Faust Csound unit generators, or opcodes, have been designed to perform these three operations in separate steps at initialisation and performance time. Faust code compilation runs at initialisation time. It does not involve any signal processing, and it does need to be repeated. More than one instance of the same Faust program can then be activated at performance time.

The performance code is subdivided into two parts, signal processing and controls, and is split into two separate opcodes. The signal processing element picks up input signals (if defined) processes them and outputs thm. Parameter control is performed by other opcodes, which can manipulate given UI components defined in the program.

Faustcompile

The `faustcompile` opcode calls the just-in-time (JIT) compiler to produce a DSP process from a Faust program. It will take a Faust program from a string. The operation can be controlled by various arguments. In Csound, multi-line strings are accepted, using {{ }} to enclose the string and S-type variables can be used to hold them.

```
ihandle faustcompile Scode, Sargs
```

`Scode` – a string containing a Faust program.
`Sargs` – a string containing Faust compiler arguments.

```
ihandle faustcompile "process=+;", "-vec -lv 1"
```

Faustaudio

The `faustaudio` opcode creates and runs a compiled Faust program. It works with code compiled with `faustcompile`:

```
ihandle,a1[,a2,...] faustaudio ifac[,ain1,...]
```

`ifac` – a handle to a compiled Faust program, produced by faustcompile.
`ihandle` – a handle to the Faust DSP instance, which can be used to access its controls with faustctl.
`ain1,...` – input signals
`a1,...` – output signals

```
ifac faustcompile "process=+;", "-vec -lv 1"
idsp,a1 faustaudio ifac,ain1,ain2
```

Faustctl

The `faustctl` opcode is used to access UI controls in a Faust DSP instance. It will set a given control in a running faust program:

faustctl idsp,Scontrol,kval

`Scontrol` – a string containing the control name
`dsp` – a handle to an existing Faust DSP instance
`kval` – value to which the control will be set.

```
idsp,a1 faustgen {{
gain = hslider("vol",1,0,1,0.01);
process = (_ * gain);
}}, ain1
faustctl idsp, "vol", 0.5
```

Faust programming is particularly useful in the prototyping and implementation of new DSP algorithms for computer instruments. Its close integration with Csound makes the three-language cooperation introduced in this chapter particularly powerful.

2.4 Programming Environment and Language Integration

At this point, it would be useful to look at how the interaction between these languages can be employed in a real-world situation. Taking Python as a host language, Csound as the audio engine, and Faust to implement a specialised audio processing algorithm, we will study the integration of these three systems introduced in this chapter. The different levels of embedding of these systems are shown in Fig. 2.2. The final section of this chapter will examine one application case study as an example of this type of integration. Note that the discussion will be steered towards the overall application design. The specific aspects of signal processing and instrument development will be explored in more detail in Part II.

2.4.1 The application

Our case study will focus on a variation of the Shapes program that is provided as one of the Python Csound API examples in the source code tree (under examples/python/shapes.py). This will be adapted to use Python 3 and the ctcsound package. Fig. 2.3 shows the main application window.

From a high-level, the program works as a synthesiser with a two-dimensional control. Once the user clicks on the circle, it changes colour from black to read and

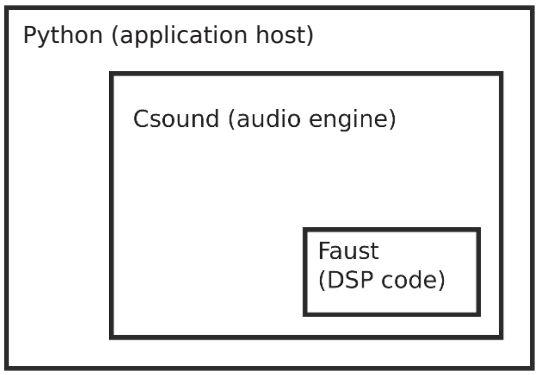

Fig. 2.2 Levels of embedding of Python, Csound, and Faust.

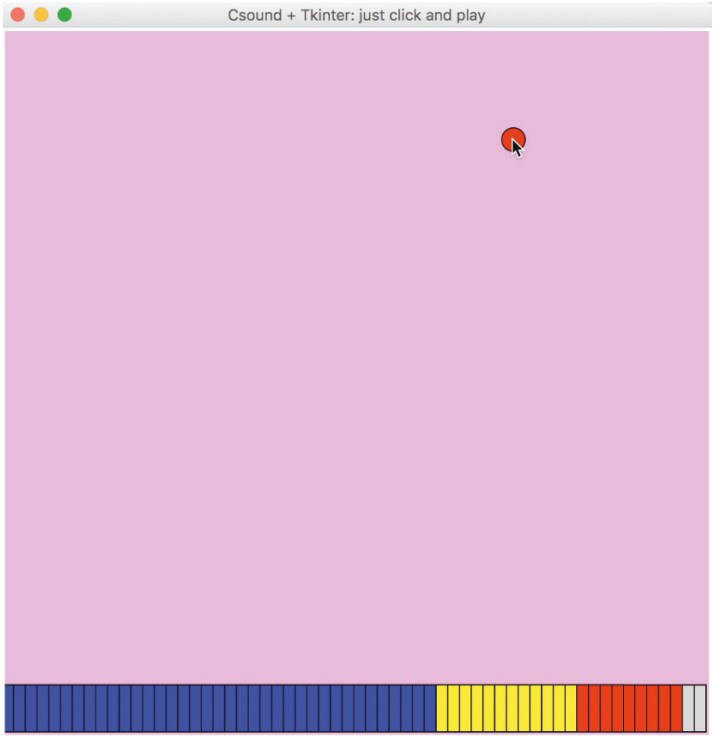

Fig. 2.3 The Shapes application.

we start hearing a sound. The graphical user interface (GUI) is designed as a canvas space to be explored by dragging the red ball around.

We can soon identify that the vertical axis controls the volume (or amplitude) of the tone, and that pitch lies in the horizontal direction. We also notice that, on clicking, there is a short *attack* followed by a long decay. When the ball is let go, the sound also continues through a shorter *release* period, in which it gets duller and softer until it stops. The bottom of the window shows a VU meter that gets illuminated when the tone is played, changing with increases in volume.

2.4.2 Instruments

The Shapes application is composed of two elements as far as its code is concerned. One of them structures the program itself, and the other controls the audio engine. The former is written in Python, while the latter is in the Csound language, containing the instrument that makes the sound we hear, which includes a process implemented in Faust.

The full Csound code for this application is contained in a Python text string:

Listing 2.3 Csound instrument code.

```
code = '''
0dbfs=1

gar init 0
gifbam2 faustcompile {{
        beta = hslider("beta", 0, 0, 2, 0.001);
        fbam2(b) = *~(((\(x).(x + x'))*b)+1);
        process = fbam2(beta);
}}, "-vec -lv 1"

instr 1
 ival = p4
 kp chnget "pitch"
 kp tonek kp, 10, 1
 kv chnget "volume"
 kv tonek kv, 10
 ain oscili 1, p5*kp, -1, 0.25
 ib, asig faustaudio gifbam2,ain
 kb1 expsegr ival,0.01,ival,27,0.001,0.2,0.001
 faustctl ib,"beta",kb1
 asig balance asig, ain
 kenv expsegr 1,20,0.001,0.2,0.001
 aenv2 linsegr 0,0.015,0,0.001,p4,0.2,p4
 aout = asig*aenv2*kenv*0.4*kv
```

```
     out  aout
 gar += aout
endin

instr 100
  a1,a2 reverbsc gar,gar,0.7,2000
  amix = (a1+a2)/2
    out amix
  ks rms gar+amix
  chnset ks, "meter"
  gar = 0
endin

schedule(100,0,-1)
'''
```

From the sounds we hear we might deduct that instrument 1 is responsible for the tone, also because the other one does not output any audio. So we can start by concentrating on that part of the code. The design of the instrument is centred around the FBAM2 algorithm[5] [50], implemented as a Faust program:

```
gifbam2 faustcompile {{
        beta = hslider("beta", 0, 0, 2, 0.001);
        fbam2(b) = *~(((\(x).(x + x'))*b)+1);
        process = fbam2(beta);
}}, "-vec -lv 1"
```

Csound produces a sinusoidal signal (a cosine wave) with its `oscili` opcode, which is fed into the FBAM2 process. The rest of the instrument is made up of envelopes, one of which will be used to control a timbral parameter in the Faust code, and two others to make a linear attack and an exponential decay. Every time the instrument is triggered, these envelopes will shape the amplitude and timbre of the output sound.

The sources for the amplitude and pitch parameters are the `"volume"` and `"pitch"` channels (the `chnget` opcode). As we have seen in Sect. 2.2.3, software bus channels such as these are used to carry control data from the Python host to Csound. On the Python side, these are accessed by the `setControlChannel()` method of the `Csound` class, as we will see in later in the code examples.

An extra effect has been added to enhance the sound in this code. In instrument 100, we have a reverb (`reverbsc`) to modify the sound generated by instrument 1. This effect instrument is scheduled to be running continuously, always on. Another purpose for it is to produce the RMS value that is in the VU display. This is sent to the Python host via another software bus channel:

```
instr 100
  a1,a2 reverbsc gar,gar,0.7,2000
```

[5] We will study this algorithm in detail later on, in Chap. 5.

```
  amix = (a1+a2)/2
    out amix
  ks rms gar+amix
  chnset ks, "meter"
  gar = 0
endin
```

Notice that we are also using a global variable (`gar`) to carry the audio signal from the source to the reverb. We need to initialise it at the top:

```
gar init 0
```

and then use it in instrument 1:

```
gar += a2
```

These ideas should give a sense of how instruments can be developed from the perspective of signal processing operations. There is scope for significant sound design work, which will be explored in the next chapters of this book.

2.4.3 Application code

In addition to DSP, we should also consider how we can interact with and drive the instruments we create. This is where Python and its hosting capabilities come into play. The Shapes program uses the Tkinter package, which is a standard GUI framework for Python, based on the Tk library:

```
import tkinter
```

Tkinter has capabilities to set up an application that will create one or more windows, place widgets (the graphical components such as buttons, sliders, text boxes, etc.), and manage user interaction with the application.

To use it, a program will have to:

1. Create a new class definition that derives from the `tkinter.Frame` base class.
2. Add any code needed to create and draw the widgets in this class.
3. Instantiate the class.

In addition, since we are embedding a Csound engine in this application, we will need to create and manage an instance of the audio engine using the API classes and methods. We can take advantage of the architecture that Tkinter imposes on the source code and place all Csound functionality in the newly created class.

Before we look into designing the application class, let's look at what we need. These are the GUI components we see in Fig. 2.3:

- A single main window frame
- A blank canvas
- A circle/ball that can be dragged around and changes colour

- A rectangle with subdivisions that change colour in response to the sound volume

Looking at the functionality that Tkinter offers, we see that the `Canvas` object might be the basic widget we are looking for. It implements an open space where structured graphics can be drawn and manipulated. So, to implement what we have described above, the application class will need to:

- Create a basic frame.
- Instantiate a Canvas.
- Draw a circle, and bind it to mouse controls.
- Draw a series of small rectangles that together look like a big one with subdivisions.

Once these things are done, we can create a Csound object and start it running. Each click on the circle will trigger a new event in instrument 1 (see Listing 2.3). We will then hear a sound, which will continue if we hold the mouse button down, and stop if we let it go.

Dragging the ball around will change the pitch and amplitude, so we will have to respond to that as well (by setting the relevant bus channels). Finally, when the application is closed, we qill have to stop the engine. This is all the application needs to implement.

Most of the work is done inside the new class we are creating. Let's call it `Application`, derive it from `tkinter.Frame`, as required, and then initialise it by following our list of tasks outlined above:

Listing 2.4 Application init code.

```
class Application(tkinter.Frame):
 def __init__(self,master=None):
  self.master = master
  self.master.title('Csound + tkinter: '
    'just click and play')
  # create main frame window
  tkinter.Frame.__init__(self,master)
  self.pack()
  # create GUI components
  self.createCanvas()
  self.createCircle()
  self.createMeter()
  # create Csound engine
  if self.createEngine() is True:
   # meter drawing
   self.drawMeter()
   # set up close button callback
   self.master.protocol('WM_DELETE_WINDOW',
                        self.quit)
   # start the application main loop
```

```
    self.master.mainloop()
    # if the engine could not start, quit
    else: self.master.quit()
```

After initialising the top-level frame, we create the GUI components, the Engine and start the application. We can look at each of these steps one at a time now. They are all methods of the `Application` class. The canvas creation method creates a square space, with a violet background, inside the main frame (its parent). For each Tk widget created, we need to call its `pack()` method for the geometry manager to place it correctly under its parent.

Listing 2.5 Canvas creation.

```
def createCanvas(self):
    self.size = 600
    self.canvas = tkinter.Canvas(self,
                                 height=self.size,
                                 width=self.size,
                                 bg="violet")
    self.canvas.pack()
```

To create a circle, we call a shape-drawing method from `Canvas` and place it on a variable (`circle`). We then bind to this circle three mouse actions: button pressing, button 1 (held) motion, and button release. Each action will trigger a callback method associated with it (which we will examine later):

Listing 2.6 Circle creation.

```
def createCircle(self):
 circle = self.canvas.create_oval(self.size/2-10,
                                  self.size/2-10,
                                  self.size/2+10,
                                  self.size/2+10,
                                  fill="black")
 self.canvas.tag_bind(circle,"<ButtonPress>",
                      self.play)
 self.canvas.tag_bind(circle,"<B1-Motion>",
                      self.move)
 self.canvas.tag_bind(circle,"<ButtonRelease>",
                      self.stop)
```

The meter is just a series of rectangles (10 of them), which are created on the canvas, and are initially coloured grey. Once Csound starts playing, the rectangles will be drawn with different colours:

Listing 2.7 Meter creation.

```
def createMeter(self):
  iw = 10
  self.vu = []
  for i in range(0, self.size, iw):
```

```
    self.vu.append(self.canvas.create_rectangle(i,
         self.size-40,i+iw,self.size,fill="grey"))
```

The final set-up method creates the engine. We instantiate a `Csound` object, compile the code, and set up two input channels for controls. Then we create a performance thread for this engine, and start it running with its `Play()` method:

Listing 2.8 Engine creation.

```
def createEngine(self):
 self.cs = ctcsound.Csound()
 self.cs.setOption('-odac')
 res = self.cs.compileOrc(code)
 if res == 0:
  self.cs.setControlChannel('pitch', 1.0)
  self.cs.setControlChannel('volume', 1.0)
  self.perf =
   ctcsound.CsoundPerformanceThread(self.cs.csound())
  self.perf.play()
  return True
 else:
  return False
```

The drawing of the meter is done at regular intervals. In order to enable this, we use the `after()` function to schedule a call to the drawing method at the end of its definition, so that it will call itself repeatedly. This method checks for the value of a meter channel sent by Csound and fills each rectangle accordingly, with different colours:

Listing 2.9 Meter drawing.

```
def drawMeter(self):
 level = self.cs.controlChannel("meter")
 level *= 16000
 cnt = 0
 red = (self.size/10)*0.8
 yellow = (self.size/10)*0.6
 for i in self.vu:
  if level > cnt*100:
    if cnt > red:
      self.canvas.itemconfigure(i, fill="red")
    elif cnt > yellow:
      self.canvas.itemconfigure(i, fill="yellow")
    else:
      self.canvas.itemconfigure(i, fill="blue")
  else:
    self.canvas.itemconfigure(i, fill="grey")
  cnt  = cnt + 1
 self.master.after(50,self.drawMeter)
```

Now we can examine the mouse action callbacks. When the mouse button is pressed, we will issue the `Play()` method, which will locate the current widget being handled by this user `event` and issue an `inputMessage()` to the engine (via its performance thread). This method takes a string with a list of parameters, starting with an identifier code ('i' for instrument event), and followed by each parameter separated by spaces:

```
'i 1 0 -1 0.5 440'
```

which should be read: instr 1, starting now (0), for an unlimited time (−1), with p4 = 0.5 and p5 = 440. As we have seen before, a negative `p3` can be used to make a sound event of indefinite duration:

Listing 2.10 Starting a sound.

```
def play(self,event):
 note = event.widget.find_withtag('current')[0]
 self.canvas.itemconfigure(note, fill='red')
 self.perf.inputMessage('i1 0 -1 0.5 440')
```

Stopping a sound uses a very similar method, except that when we issue an event to Csound, we need to send a negative `p1`:

Listing 2.11 Stopping a sound.

```
def stop(self,event):
 note = event.widget.find_withtag('current')[0]
 self.canvas.itemconfigure(note, fill='red')
 self.perf.inputMessage('i-1 0 1 0.5 440')
```

When moving the mouse to drag a circle, we will need to do two things: (a) control Csound and (b) move the circle by tracking the mouse position. The canvas is the widget to which the user event is registered. We can get the canvas coordinates of the event, and as before get the item (circle) we are manipulating. By calling the `coords` method of the `Canvas` class, we can move the item anywhere inside the canvas. Finally, we just set the control channels in the Csound engine according to the position of the circle:

Listing 2.12 Moving the circle.

```
def move(self,event):
  canvas = event.widget
  x = canvas.canvasx(event.x)
  y = canvas.canvasy(event.y)
  item = canvas.find_withtag("current")[0]
  canvas.coords(item, x+10, y+10, x-10, y-10)
  self.cs.setControlChannel("pitch", 2.0*x/self.size)
  self.cs.setControlChannel("volume",
                      2.0*(self.size-y)/self.size)
```

To complete the class, we need to implement a `quit()` method that will stop the performance thread, so that Csound can exit cleanly. To do this, we stop the thread,

which will also close the engine, and join it until it finishes, and then we destroy the GUI objects:

Listing 2.13 Cleaning up.

```
def move(self,event):
def quit(self):
  self.perf.stop()
  self.perf.join()
  self.master.destroy()
```

The main program then becomes just a matter of instantiating the application class:

```
Application(tkinter.Tk())
```

The full Python code for this program can be found in Appendix B (Listing B.1). Further details on the development of user interfaces will be explored in Chap. 8.

2.5 Conclusions

In this chapter, we introduced the three programming languages that will be used throughout this book. We discussed the most salient details, exploring aspects of each language that will be relevant in the next chapters. It was demonstrated how these three systems can be integrated into a smooth working environment for instrument development.

It is true that Python, Csound and Faust could potentially be used on their own for this purpose. In fact, in this book we will be employing them separately at certain times. In particular, we will also use Python to describe and demonstrate aspects of signal processing that are relevant to our discussion.

However, it is good to be able to point out the advantages of a multi-language approach to programming. Each one of the systems introduced here has a particular strength: Python with its simple syntax and wide range of facilities for application hosting; Csound, as a leading DSL for music and audio; and Faust with its support for efficient implementation of new DSP algorithms. The combination of these qualities provides for a very rich computer music experience.

http://www.springer.com/978-3-319-63503-3

Computer Music Instruments
Foundations, Design and Development
Lazzarini, V.
2017, XX, 361 p. 146 illus., 43 illus. in color., Hardcover
ISBN: 978-3-319-63503-3