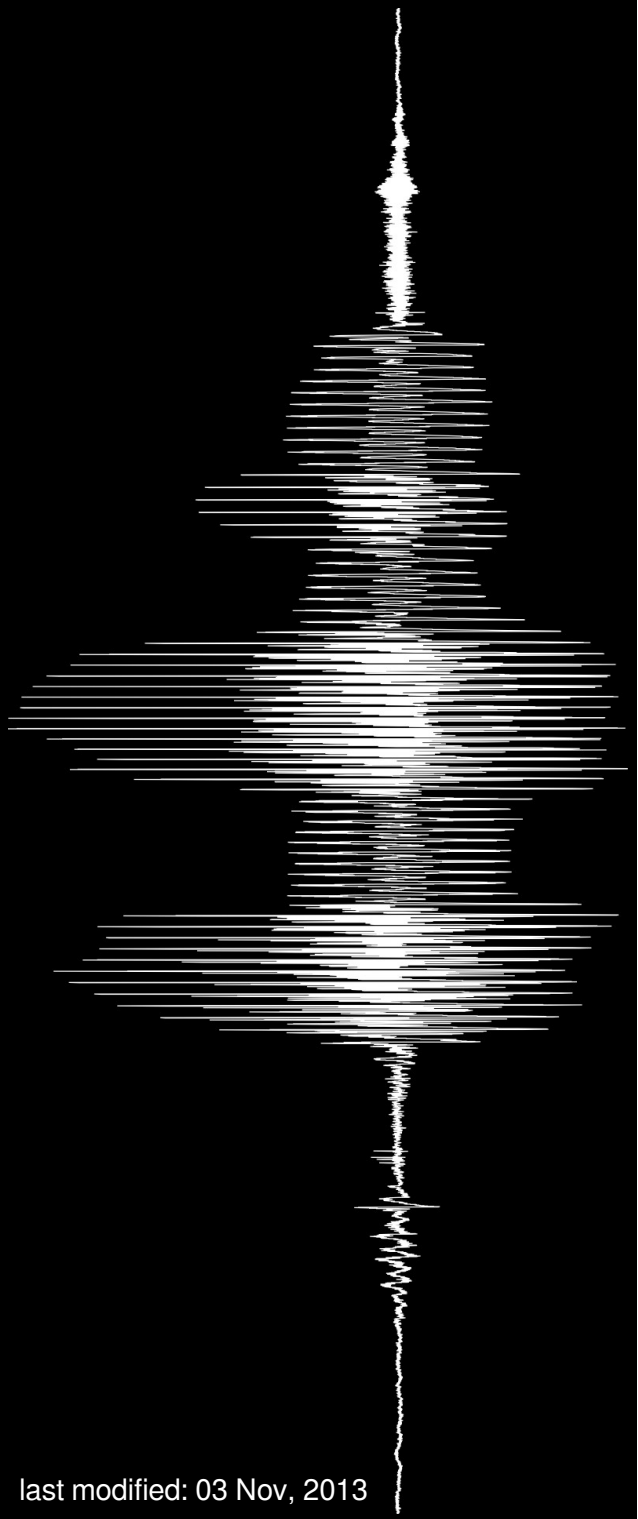
praat scripting primer



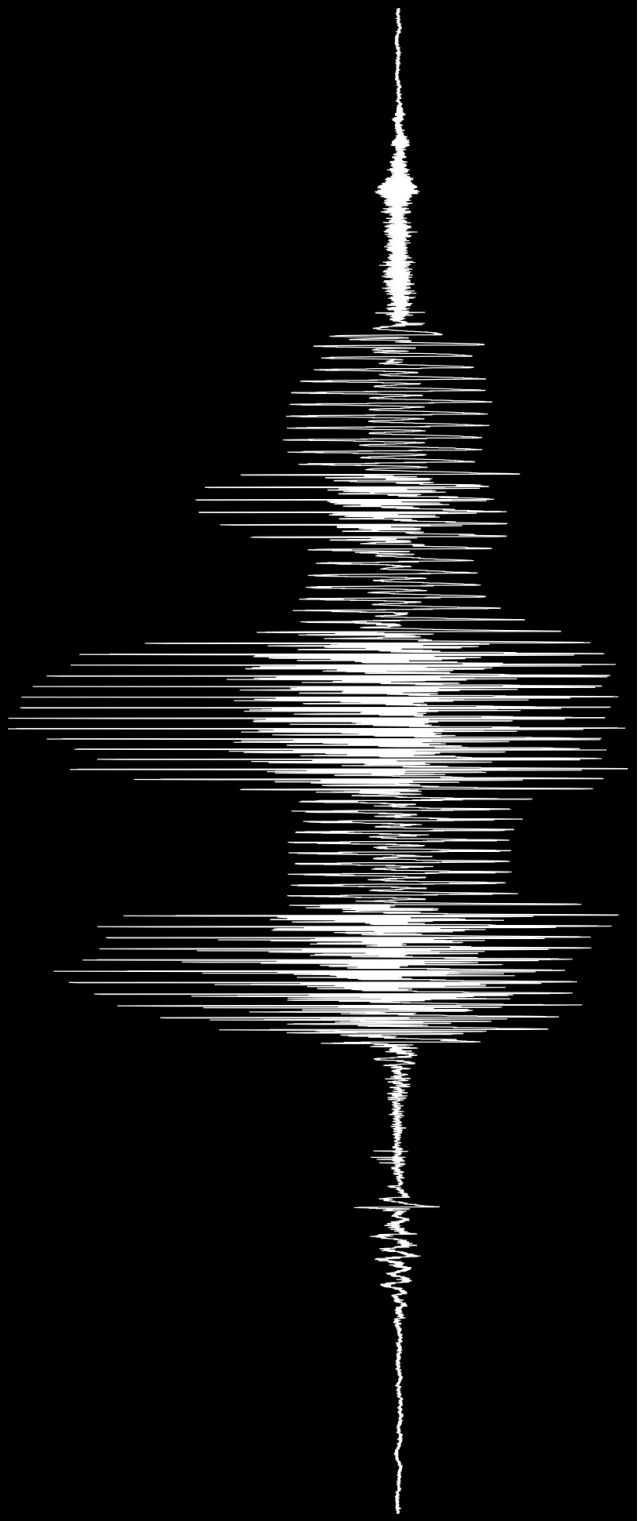
diving in head first

José Joaquín ATRIA University College London
j.atria.11@ucl.ac.uk www.pinguinorodriguez.cl

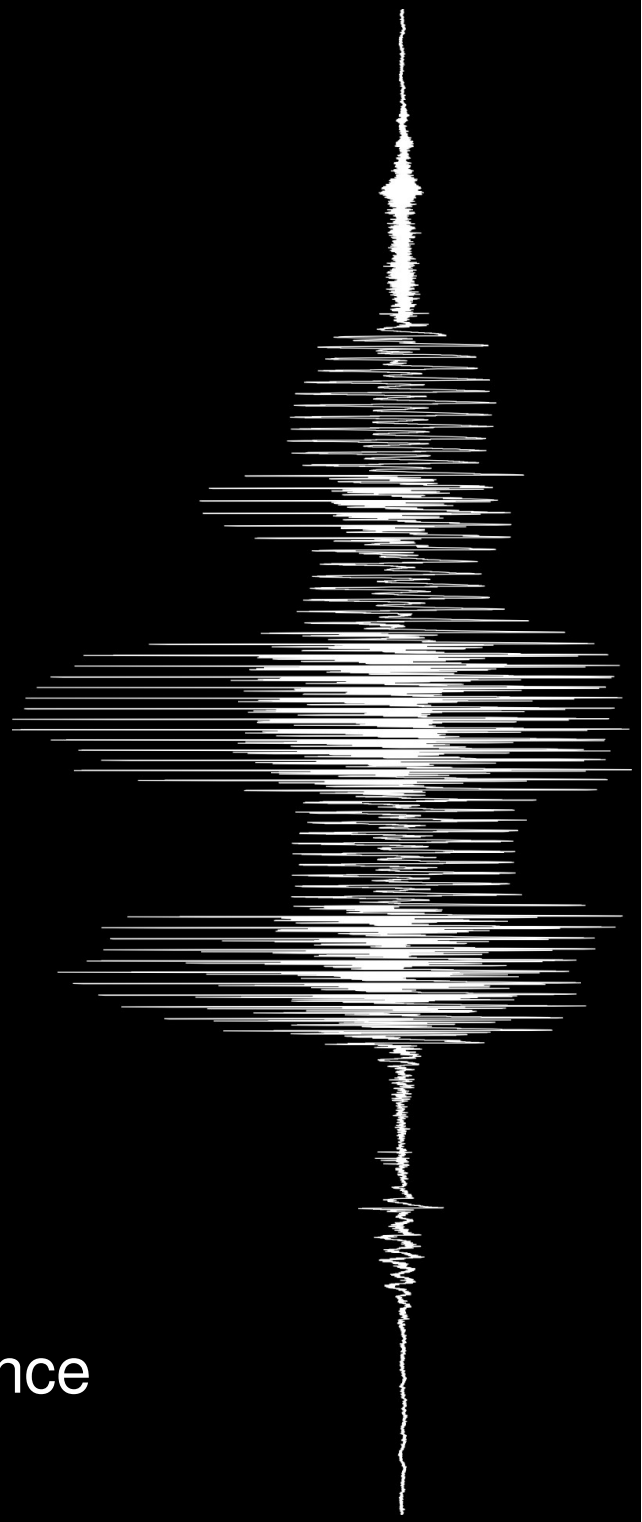
last modified: 03 Nov, 2013

part 1

overview



why use *scripts*?



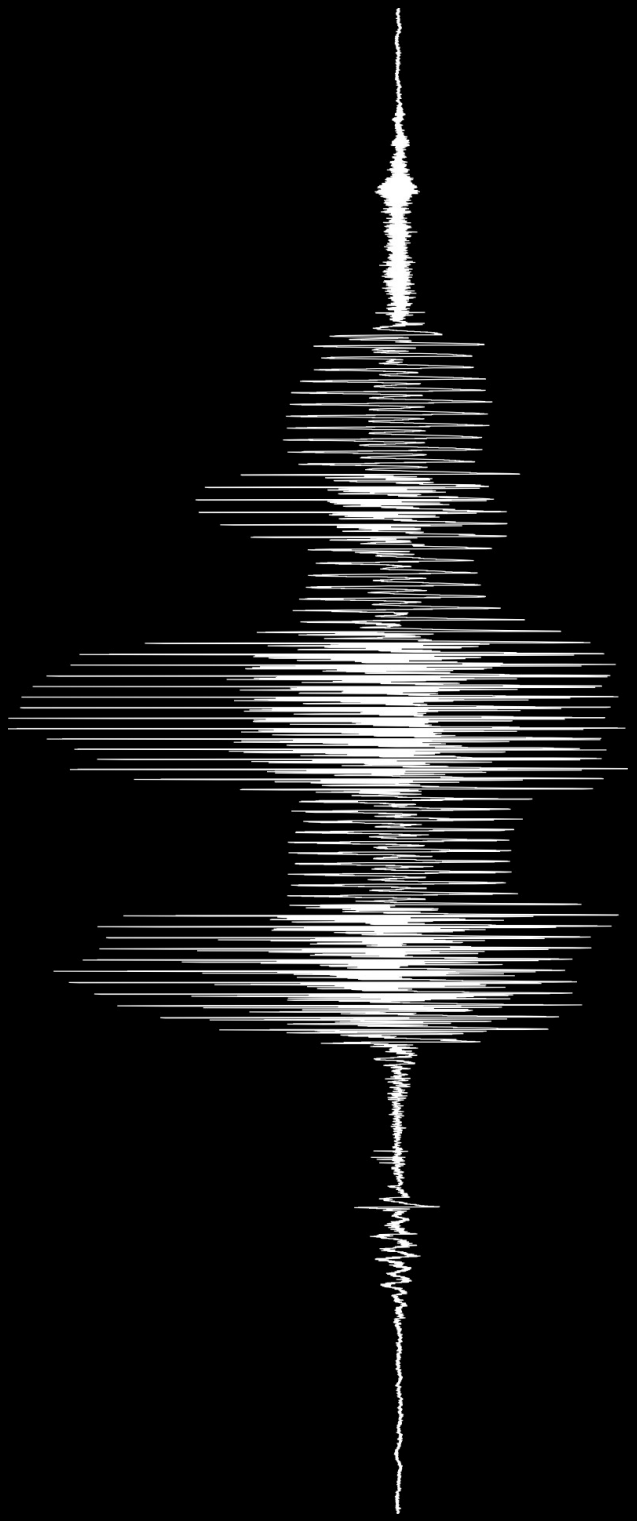
they are:

- precise
- reusable
- automatic
- portable

... which means they are the best friends of science

what is a *script*?

- a set of instructions
- stored as a separate text file
- loaded and run from within praat

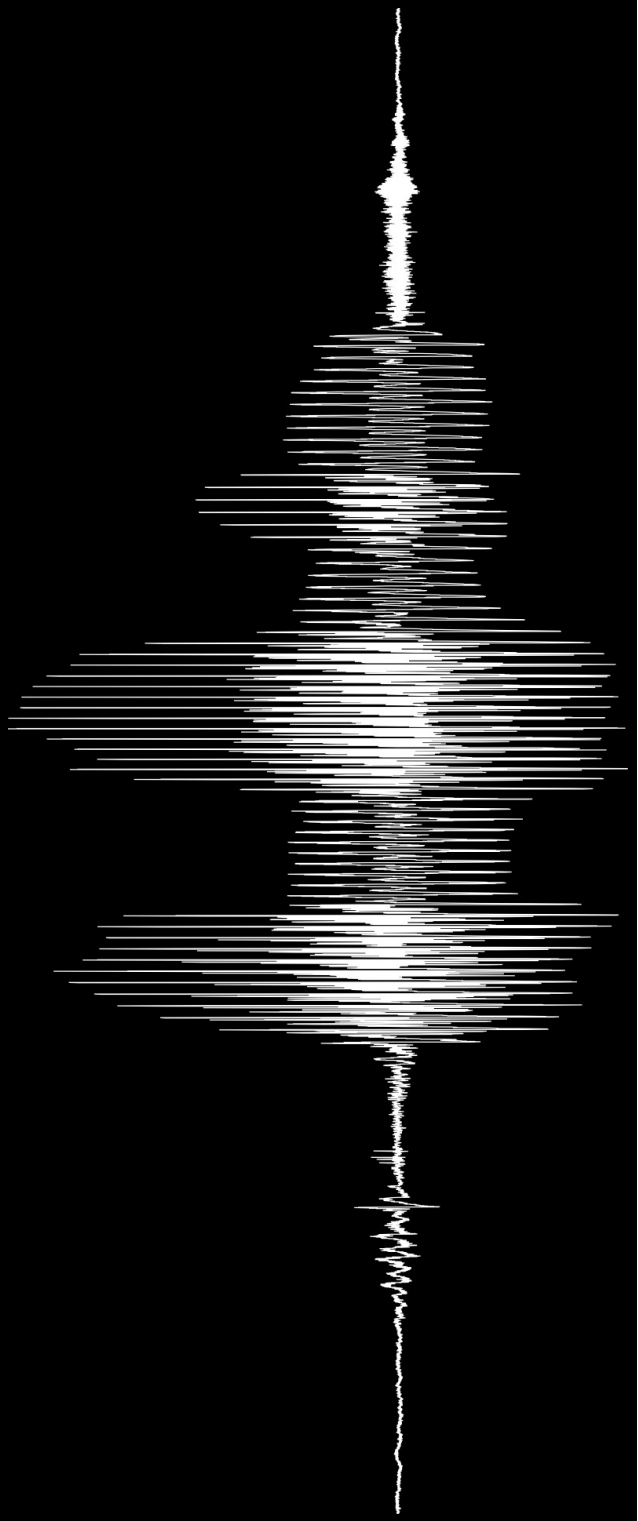


what can you do with a script?

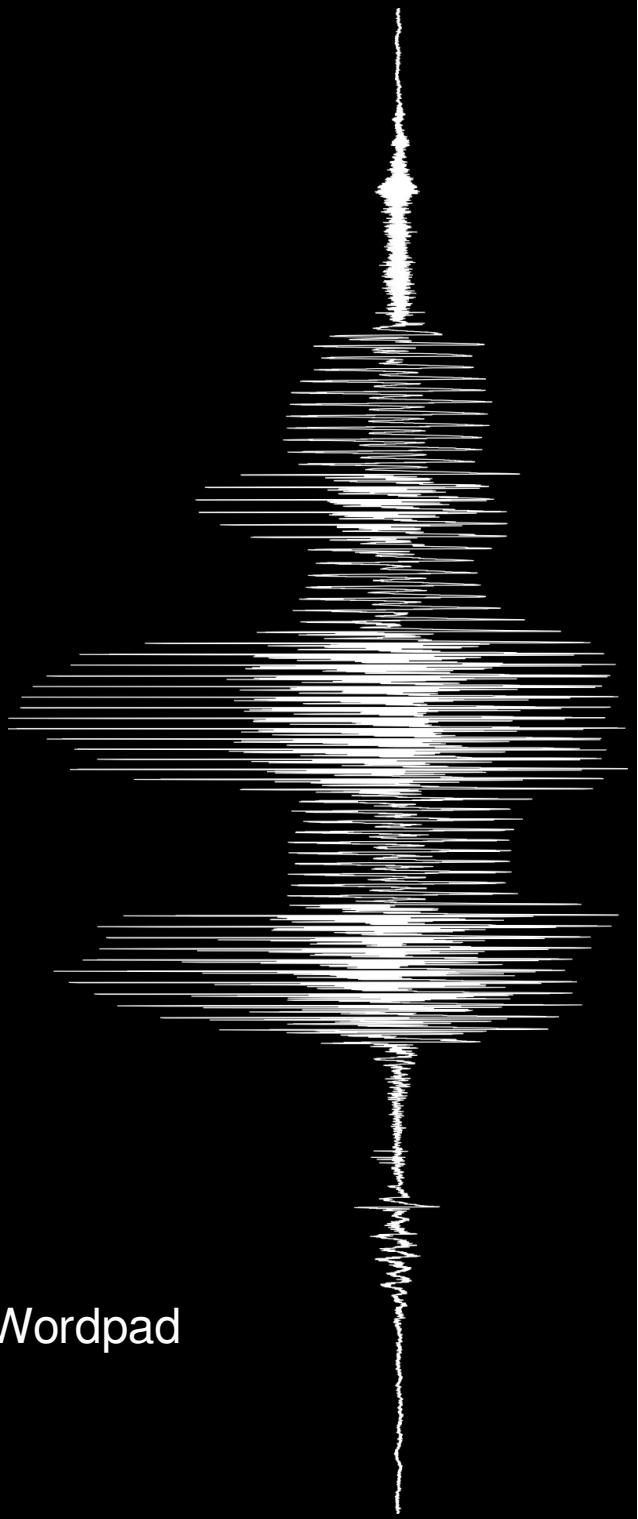
everything you can do without one

... and more:

- automatize tedious tasks
- invoke other scripts
- modify the behaviour of praat
- etc...



how do you make one?



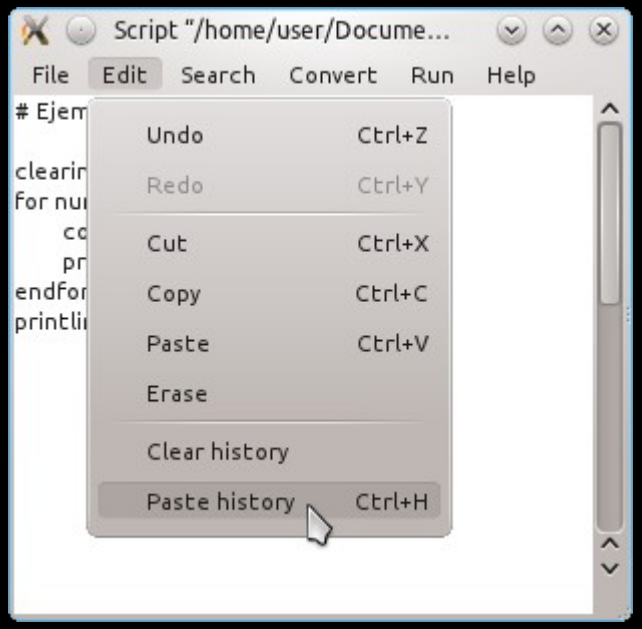
with a text editor*

- Notepad++ (Windows)
- Kate on (KDE) or Geany on (GTK)
- TextWrangler (Mac)
- praat includes its own internal editor

* i.e. ***not*** a word processor; think Notepad, not Microsoft Word or Wordpad

how do you make one?

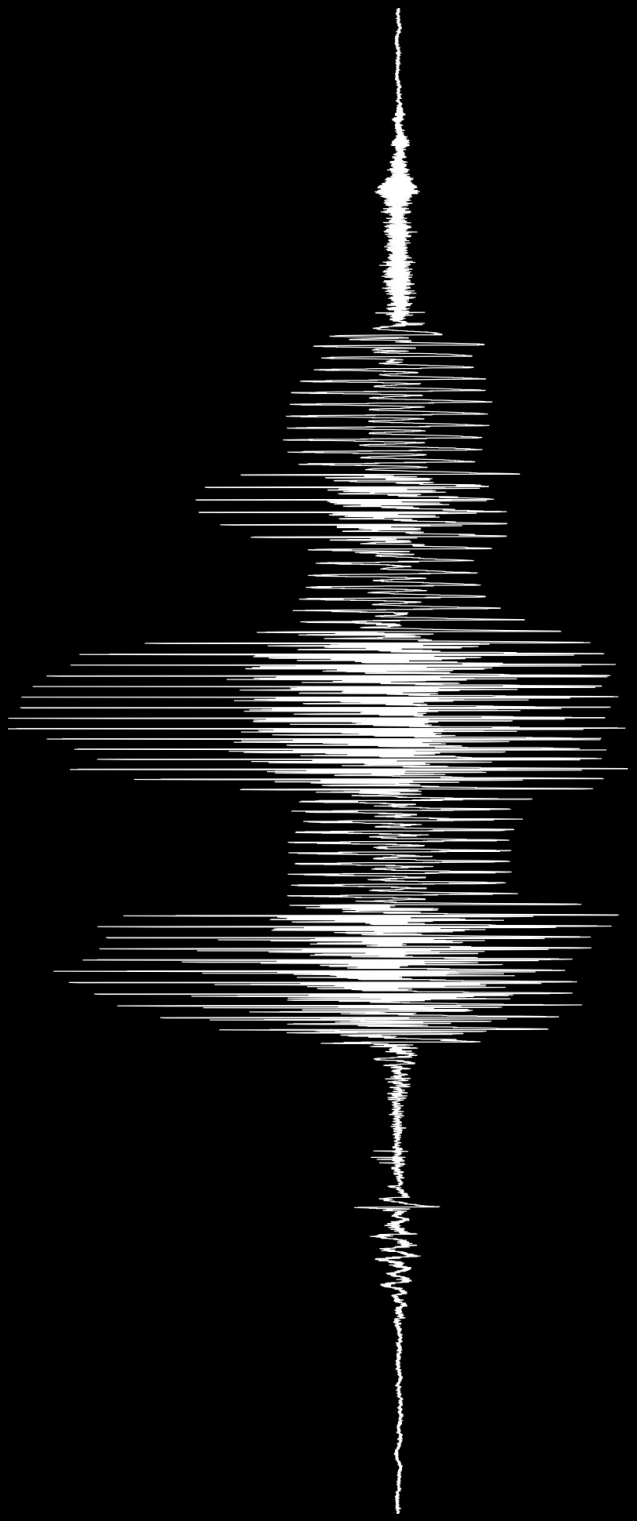
- use praat's history



with it, you sometimes need only to make a few basic modifications

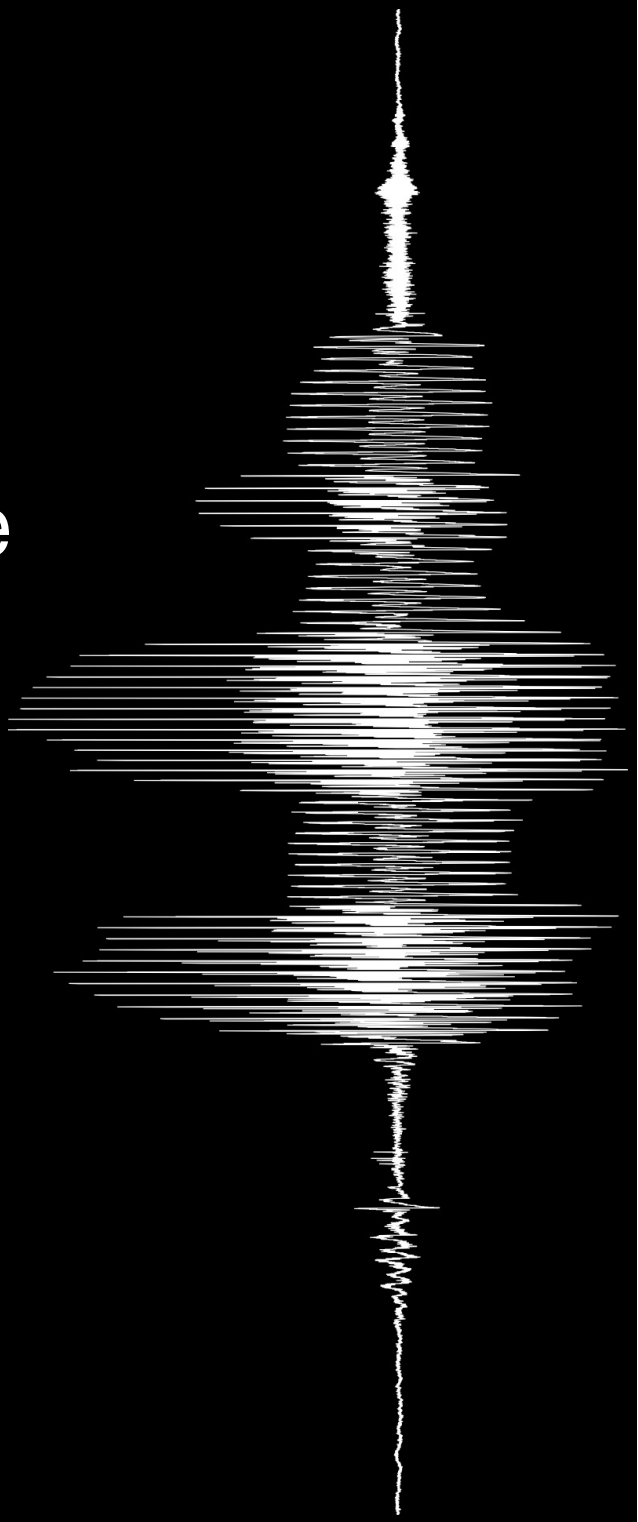
but remember:

a script is a set of instructions

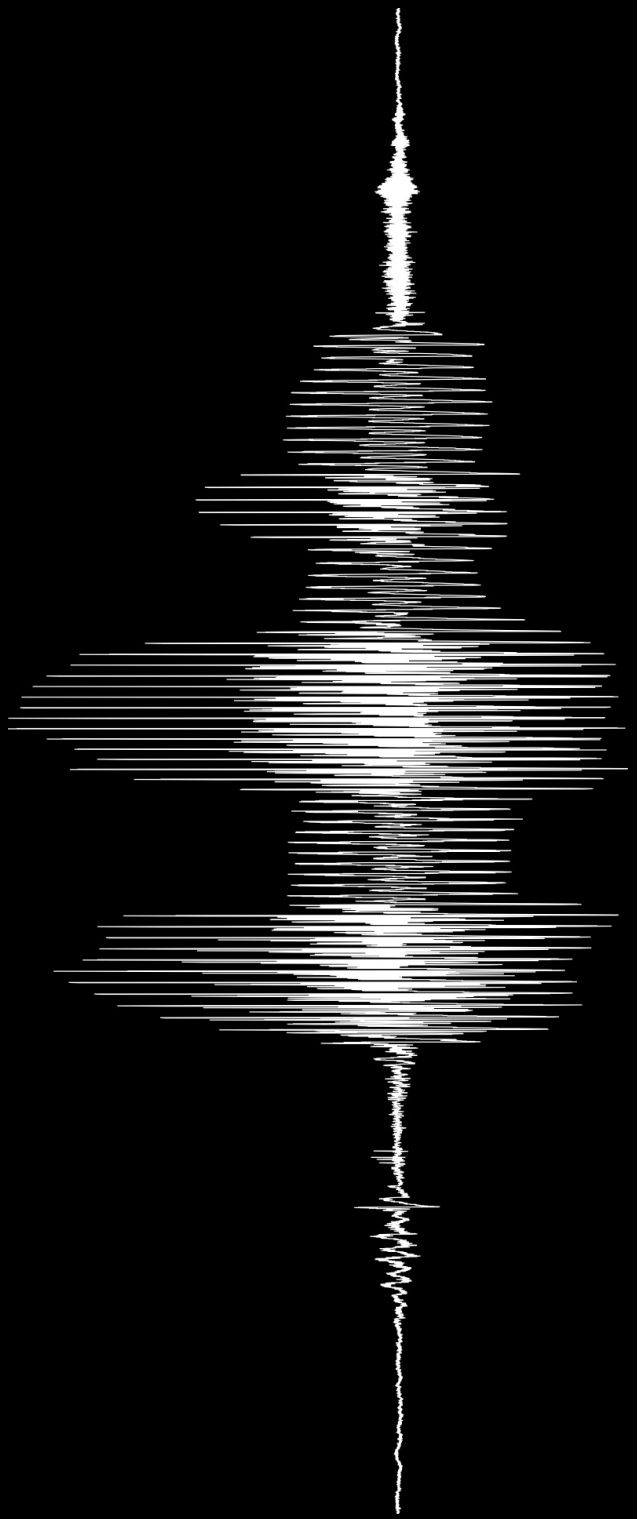


the single most important ability when writing any sort of code is the ability to take a complex problem and break it down into a sequence of simple tasks

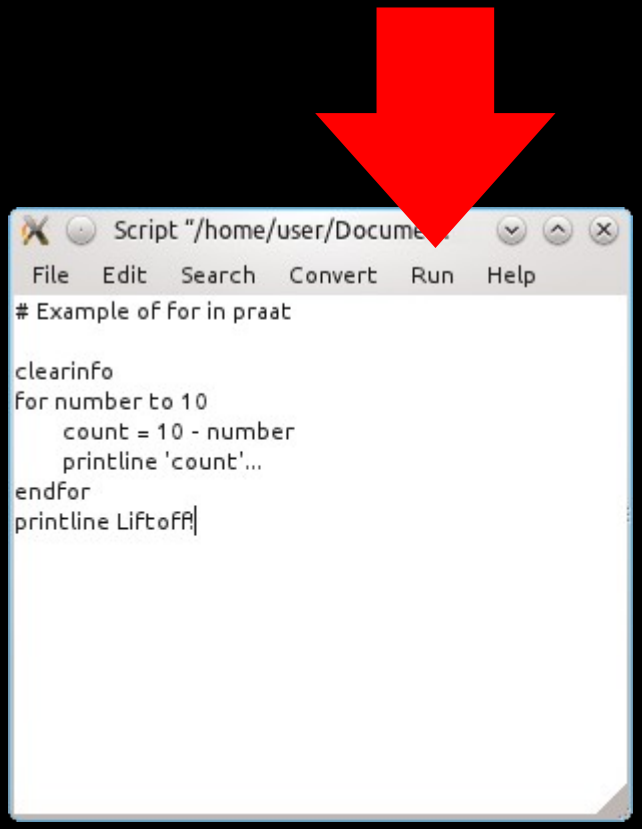
writing a script is solving a puzzle



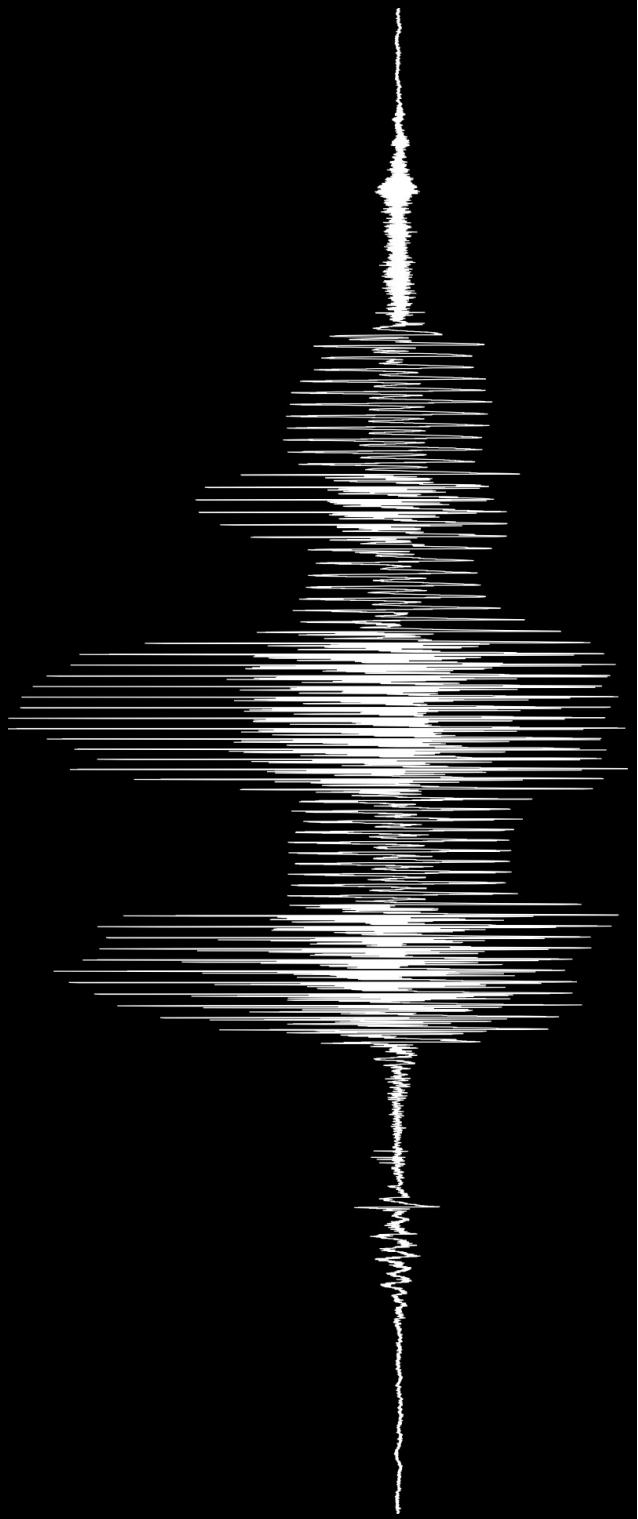
how do you use a script?



how do you use a script?

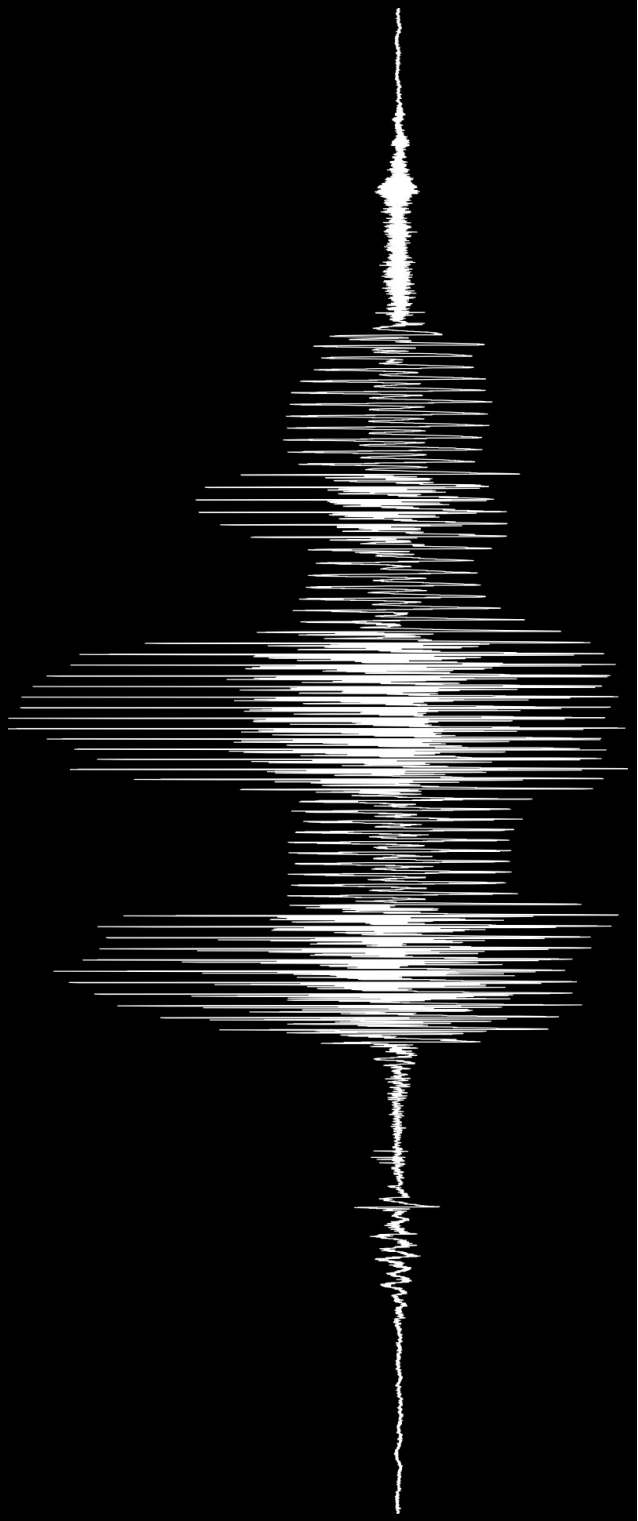


or press CTRL + R / ⌘ + R



part 2

doing it right

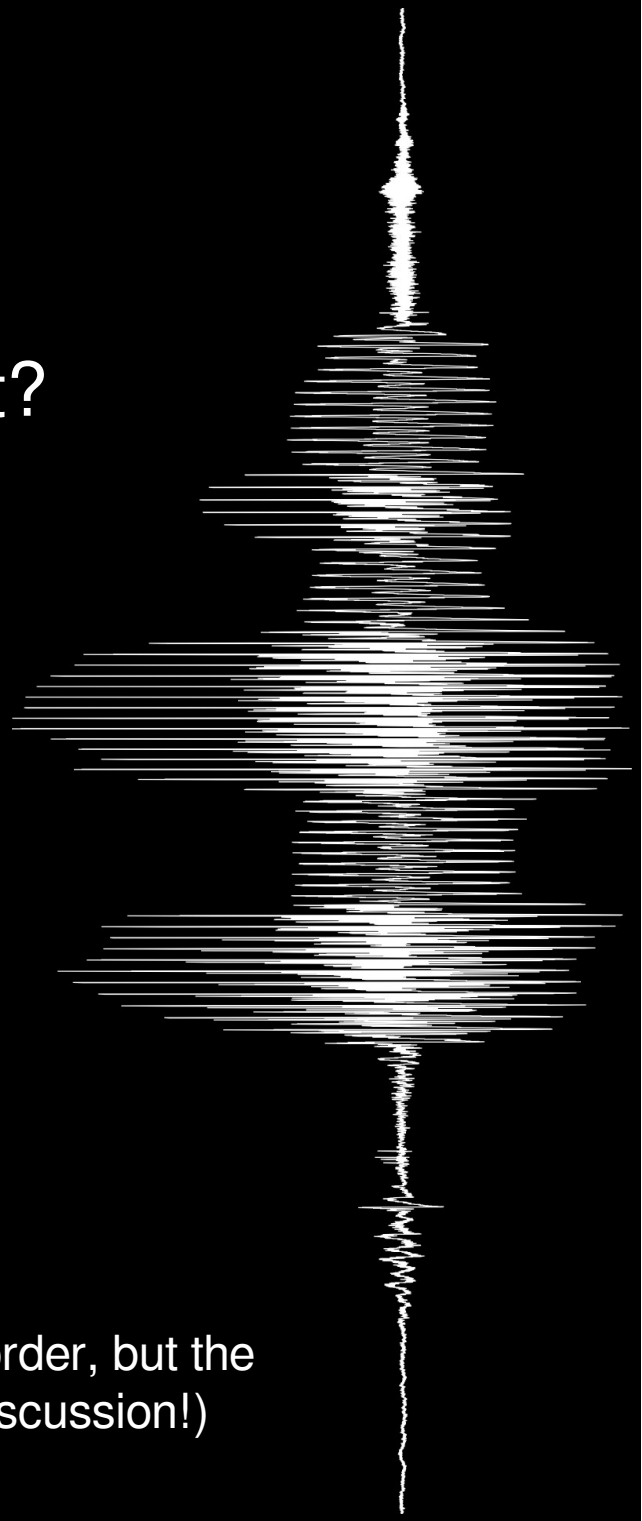


good practices

what distinguishes a good and a bad script?
in decreasing order of importance:

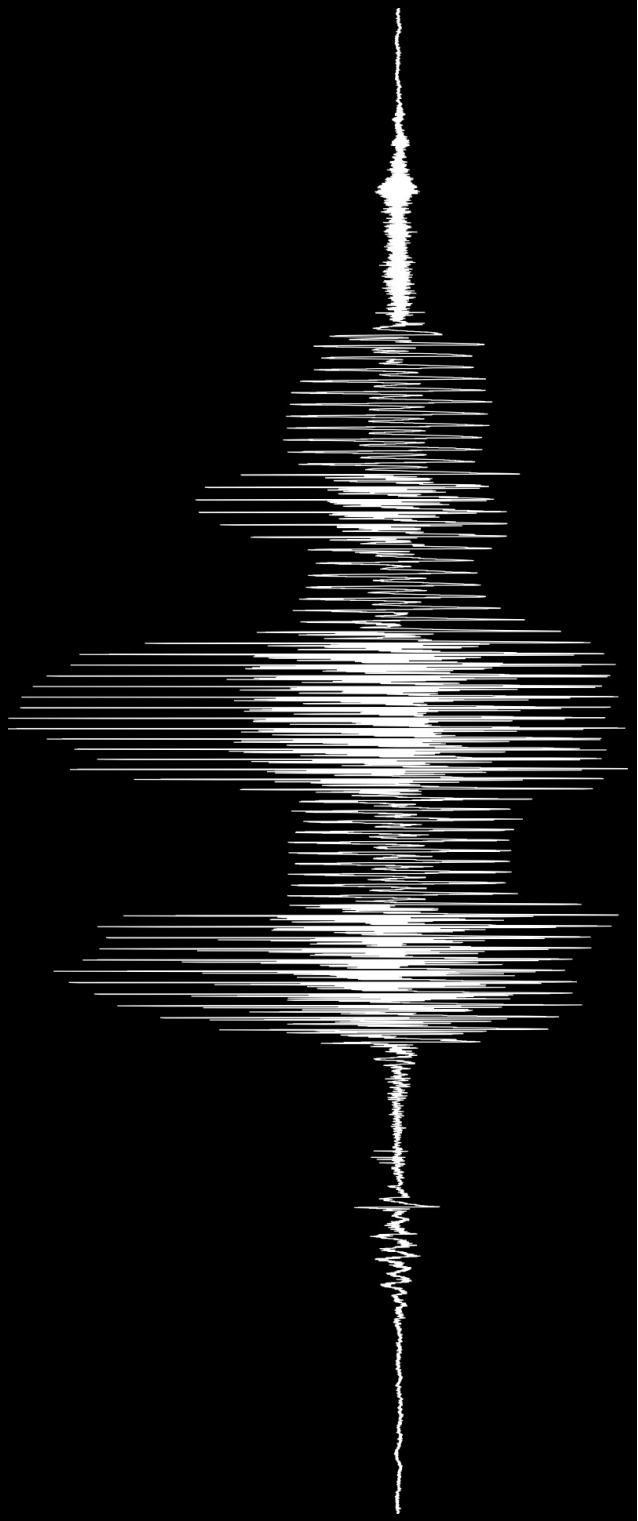
- is it easy to read?
- is it clear?
- is it extensible?
- does it work?
- is it robust?
- is it efficient?

(we may disagree on the order, but the questions are not up for discussion!)



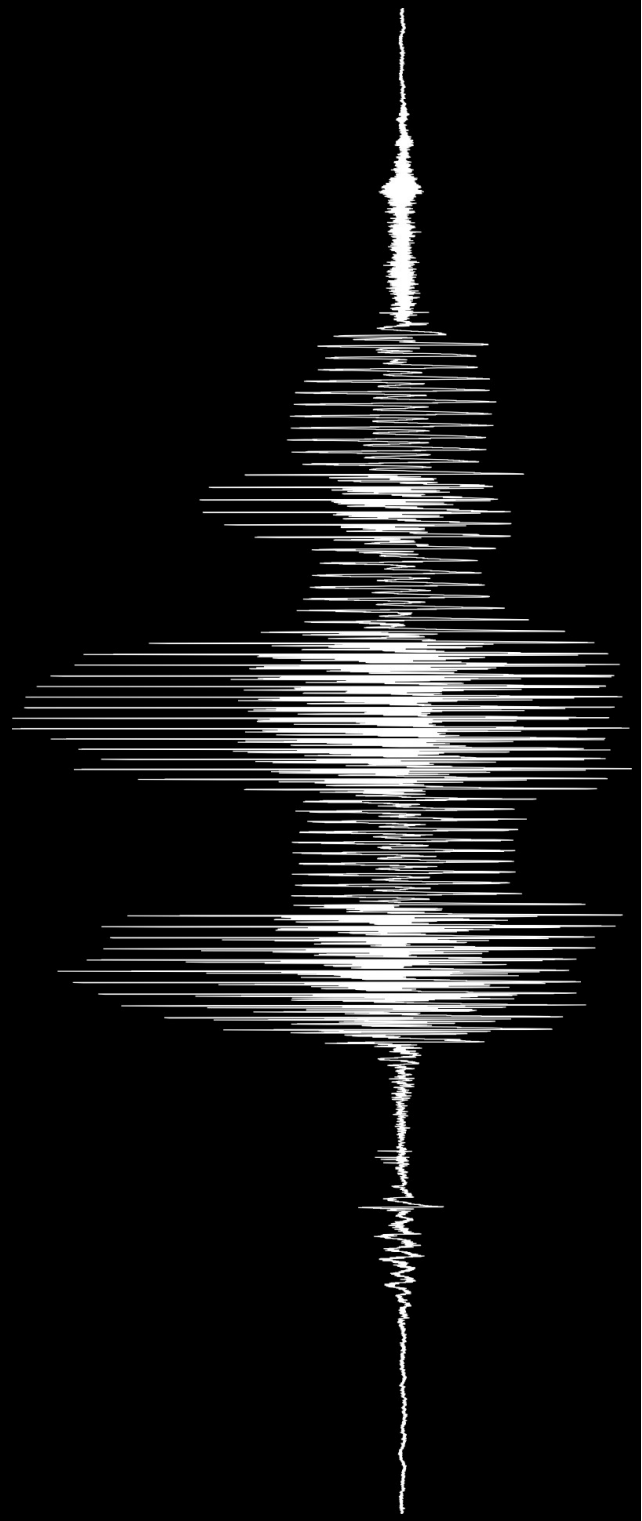
is it easy to read?

- does it have a consistent style?
- is it properly indented?
- are variable names informative?
- is it thoroughly commented?
- is it easily understood by others?
- is it possible (for you or others) to go back to it in a couple of months and not die trying?



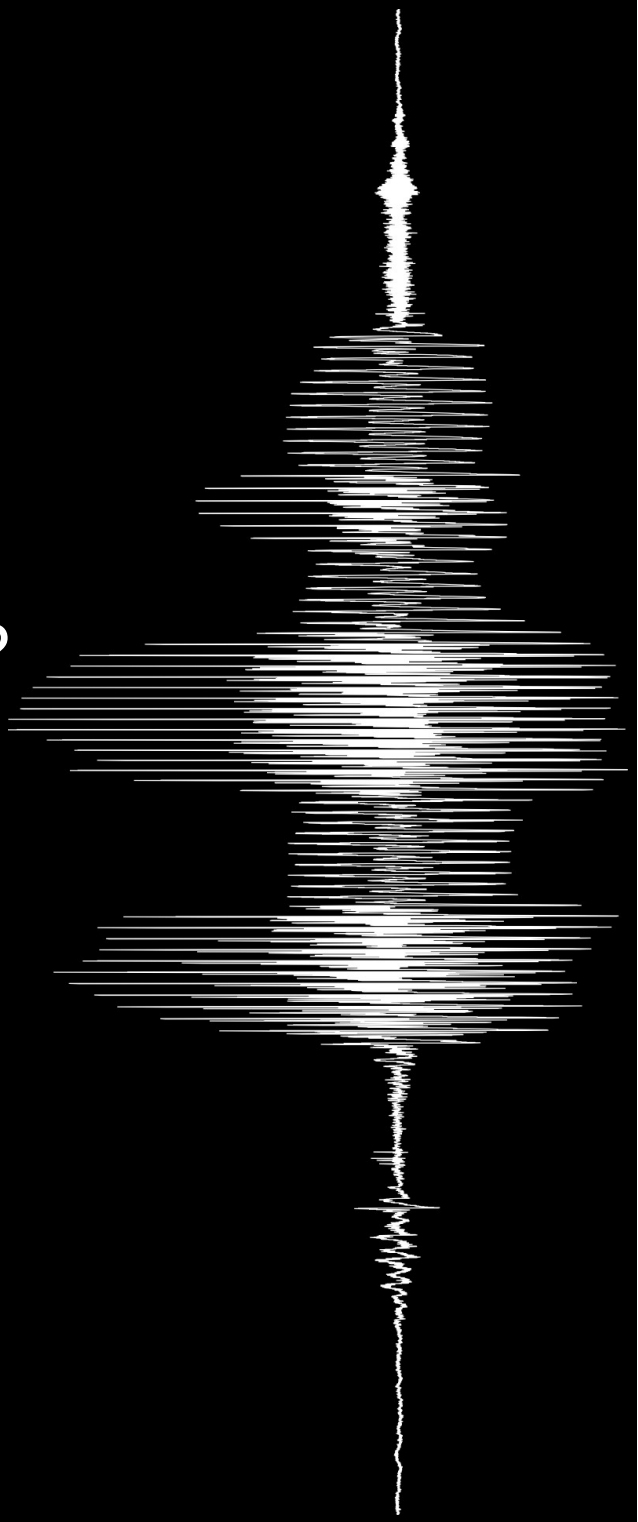
is it clear?

- does it have clear objectives?
- is it transparent in the ways to achieve them?
- is it well structured?



is it extensible?

- is it possible to use it in other situations?
- how heavily do you have to modify it to use for similar tasks?
- can it be used in different ways? or from different environments?

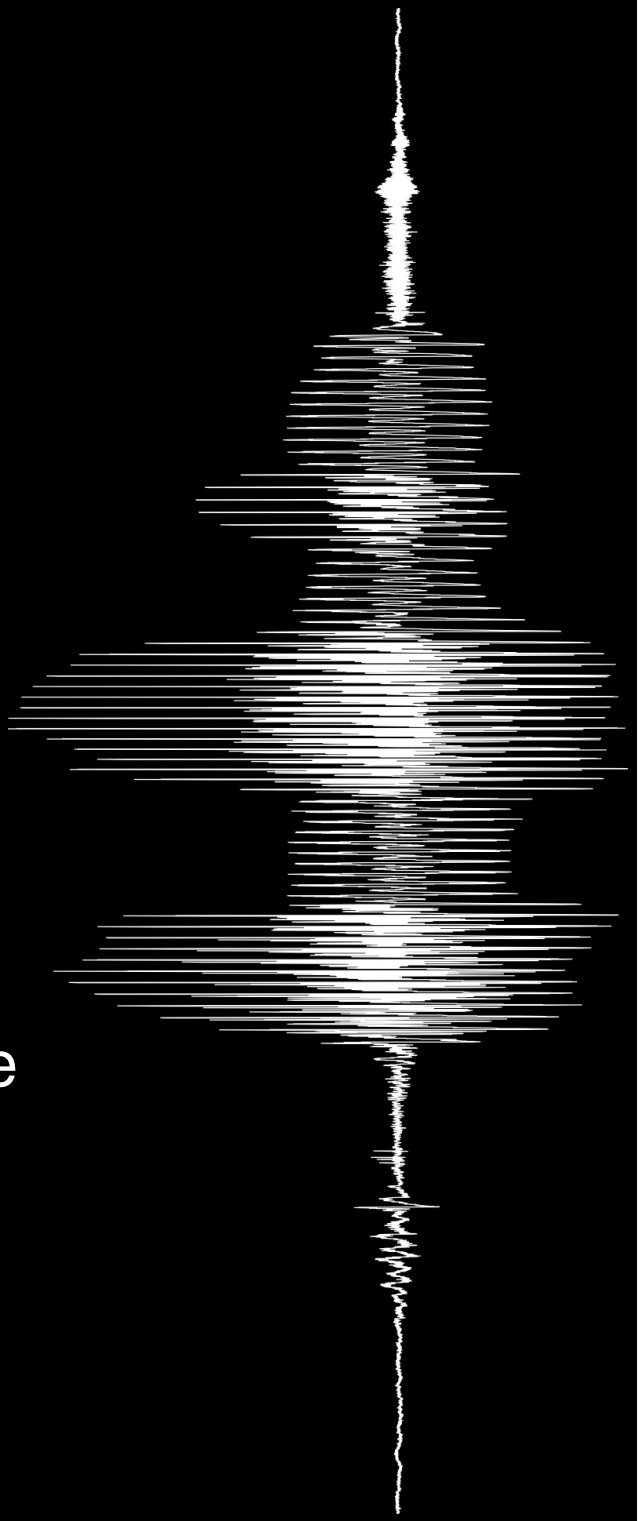


does it work?

- does it do what it has to do?

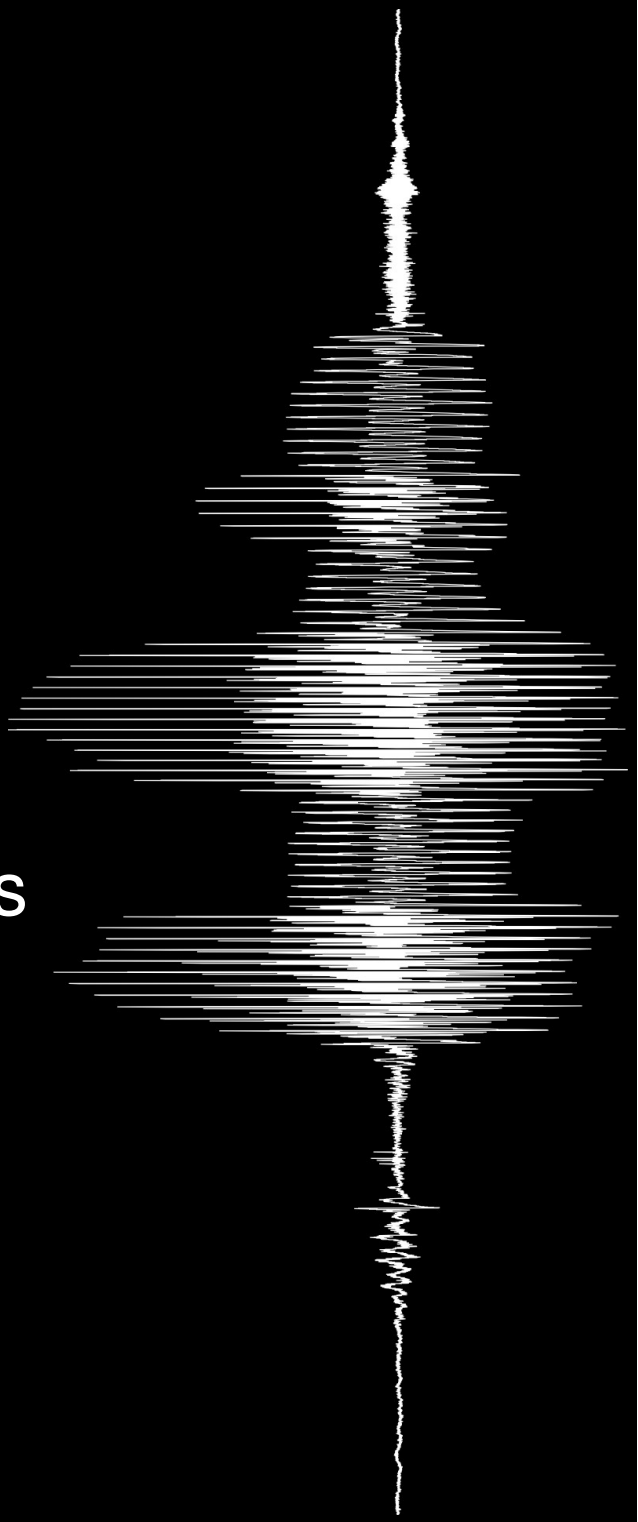
  an easily understood and well structured script that doesn't work can be fixed rather easily

  one that works in cryptic ways like a black box will soon break down, and then it will be completely useless



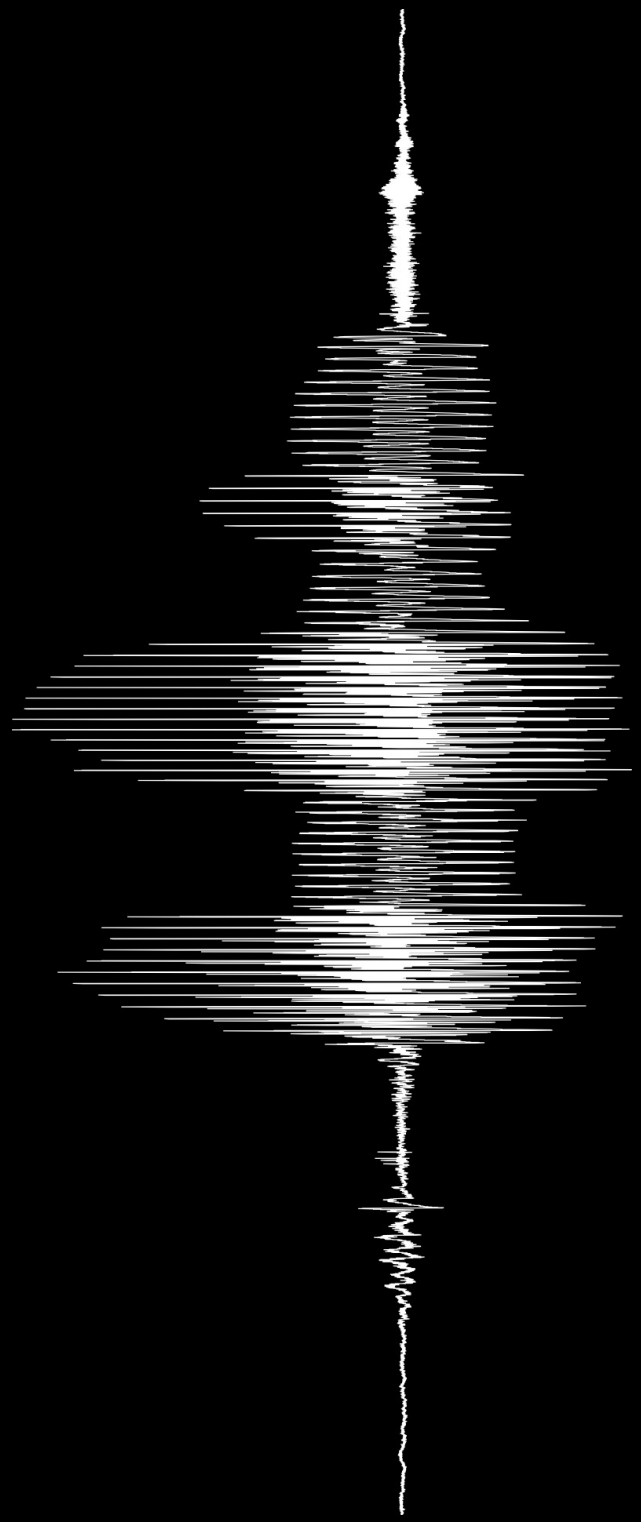
is it robust?

- how well does it react to user errors?
- how does it fare when the environment is not what is expected?



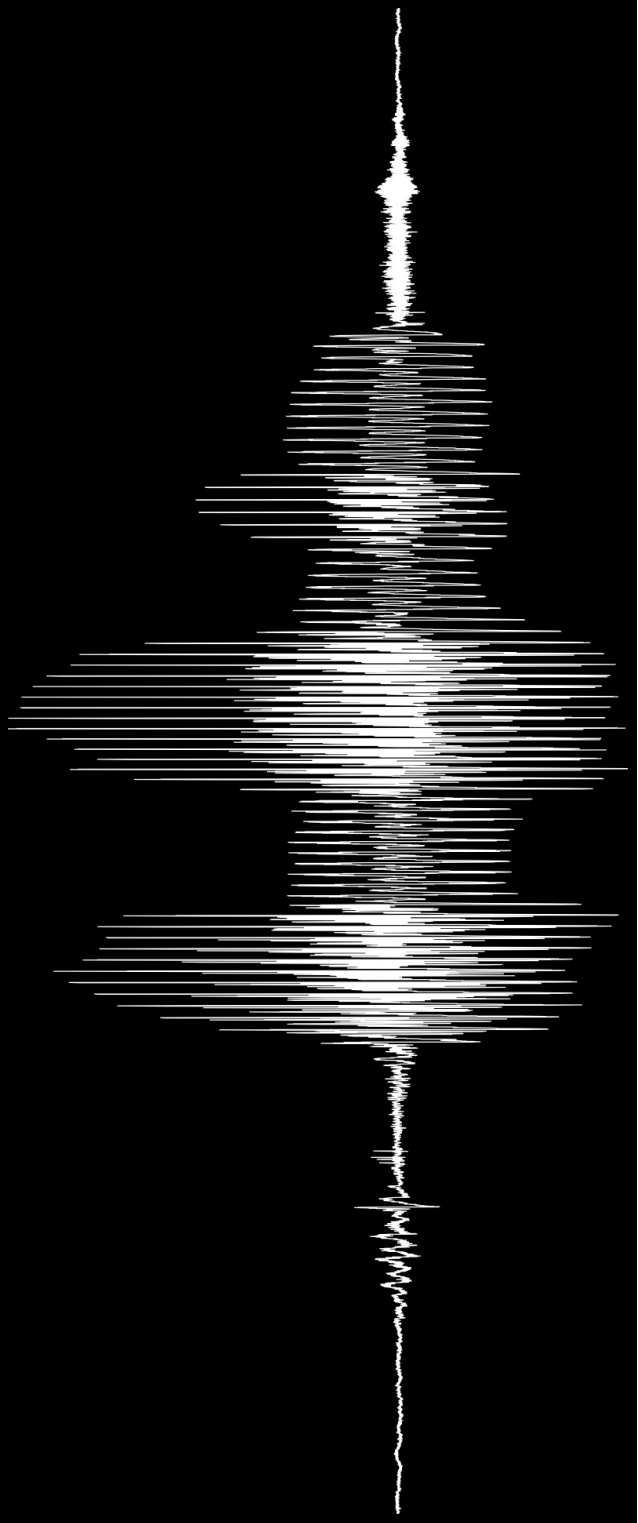
is it efficient?

- are the methods it uses the most appropriate?
- are they the fastest?
- are they the most cost-efficient in terms of memory or processor use?



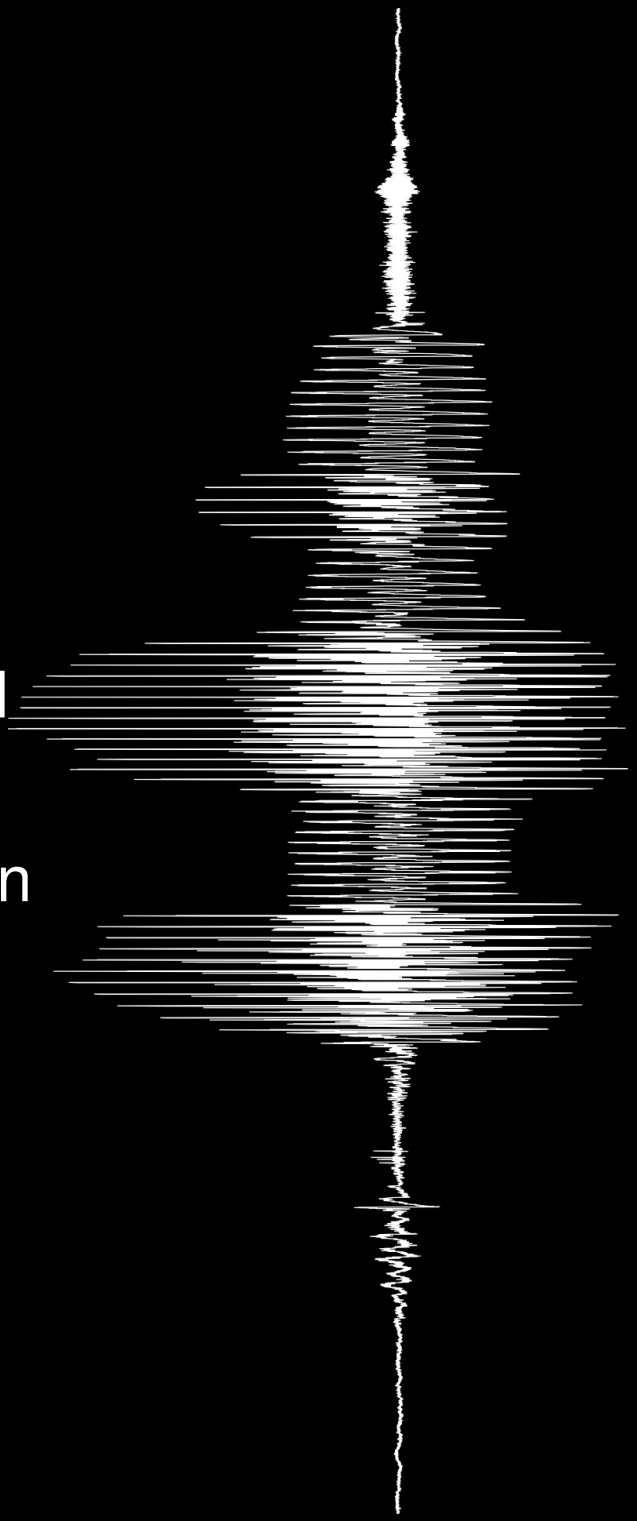
part 3

praat's scripting language



praat's scripting language

- objects
  - almost any action in praat results in the creation of an object*
  - they can be created, manipulated, removed and saved to disk with scripts
  - they last for the duration of the praat session

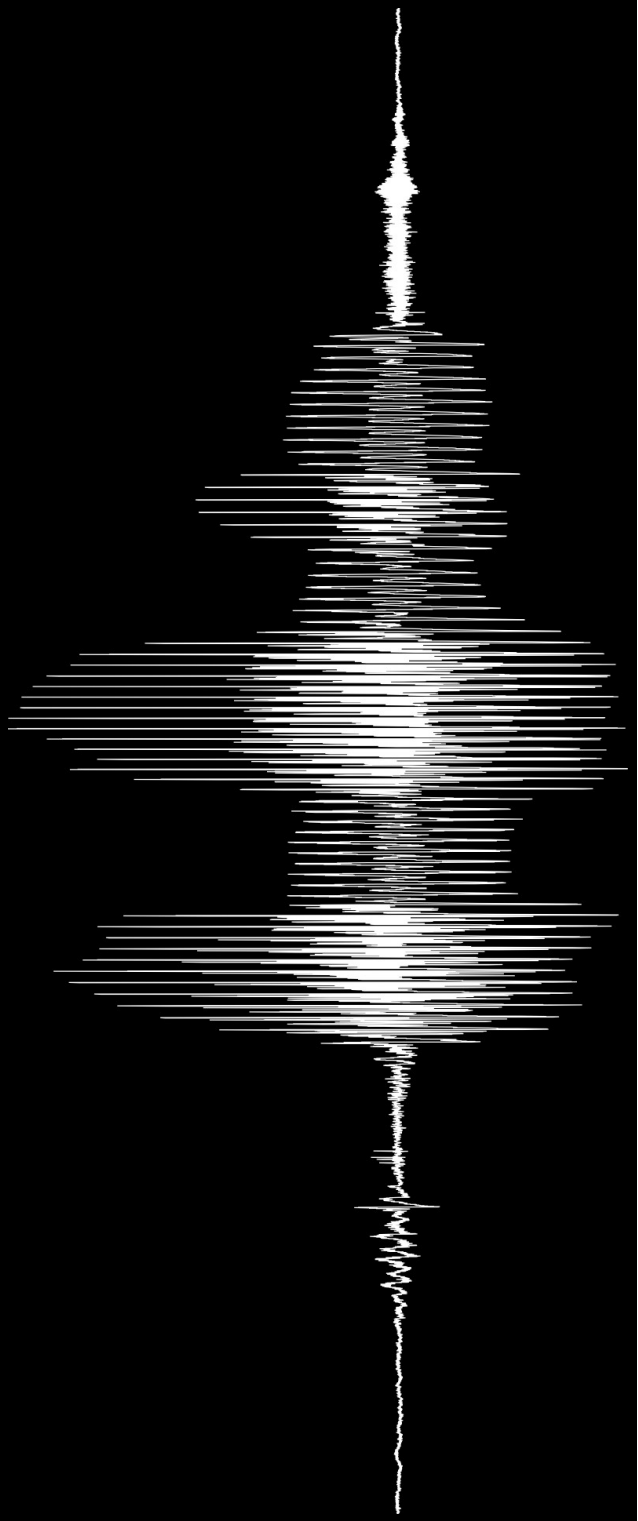


* *almost* means *not all*

praat's scripting language

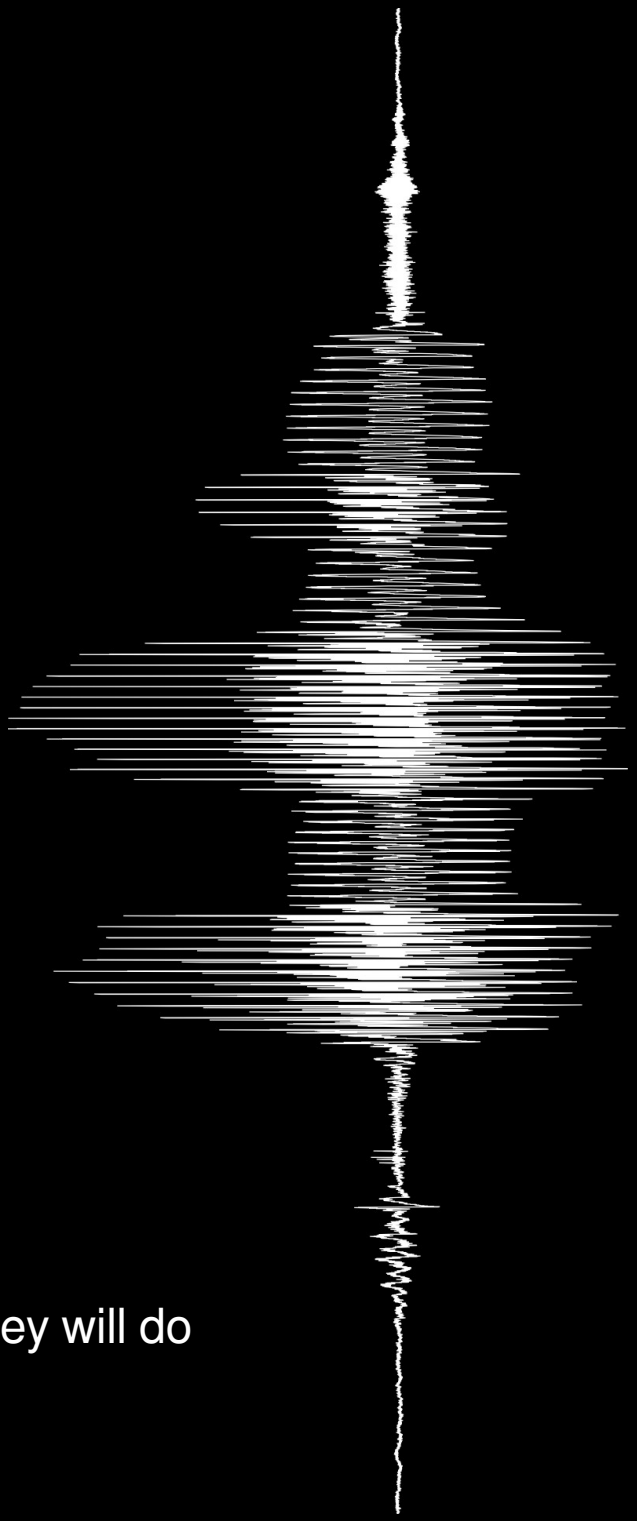
- objects
  - Sound
  - TextGrid
  - Pitch
  - Table

  and oh so many more...



praat's scripting language

- variables
  - `strings$ = "this is a string"`
  - `numeric_variables = 1337`
  - `arrays[]`*
- assignments
  - like the first two above
  - by means of queries

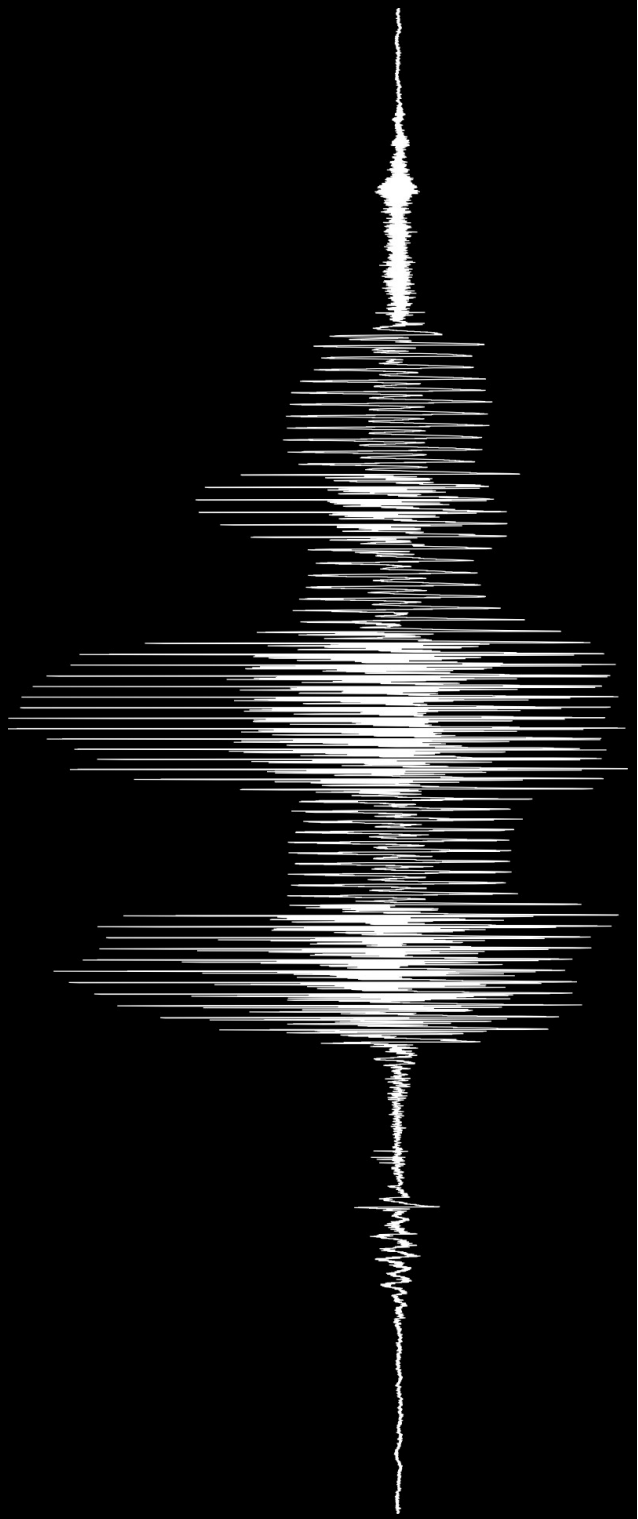


* for a number of reasons, praat's arrays are not real arrays, but they will do

praat's scripting language

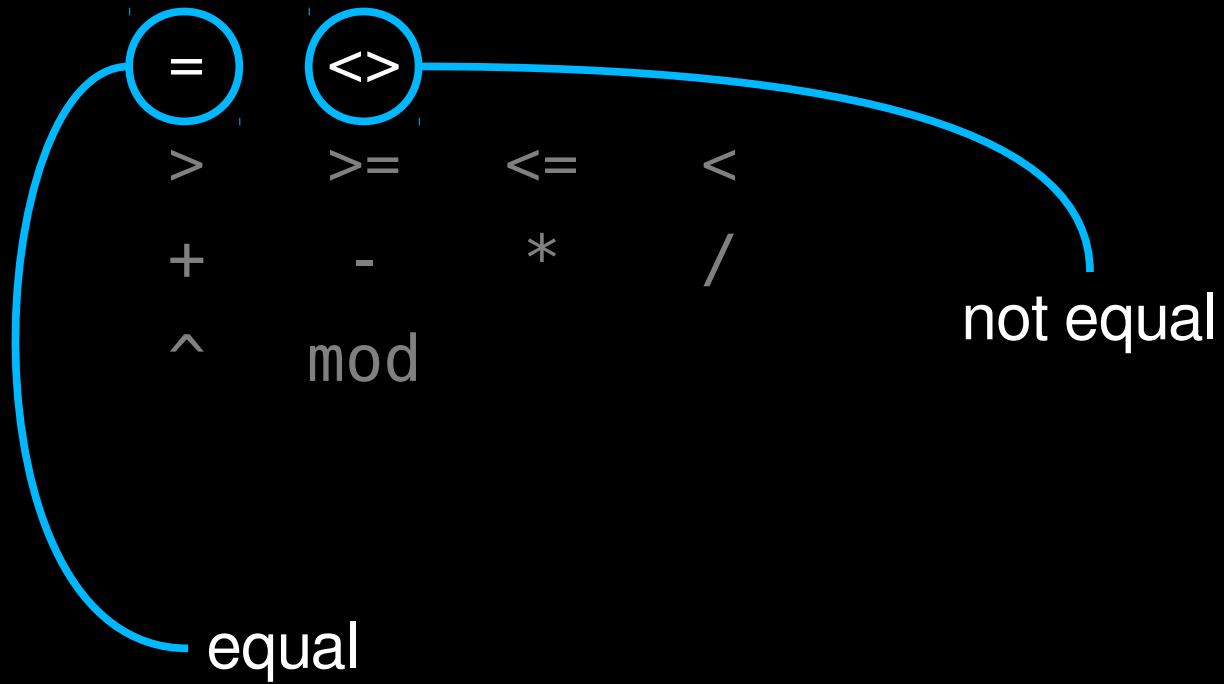
- operators

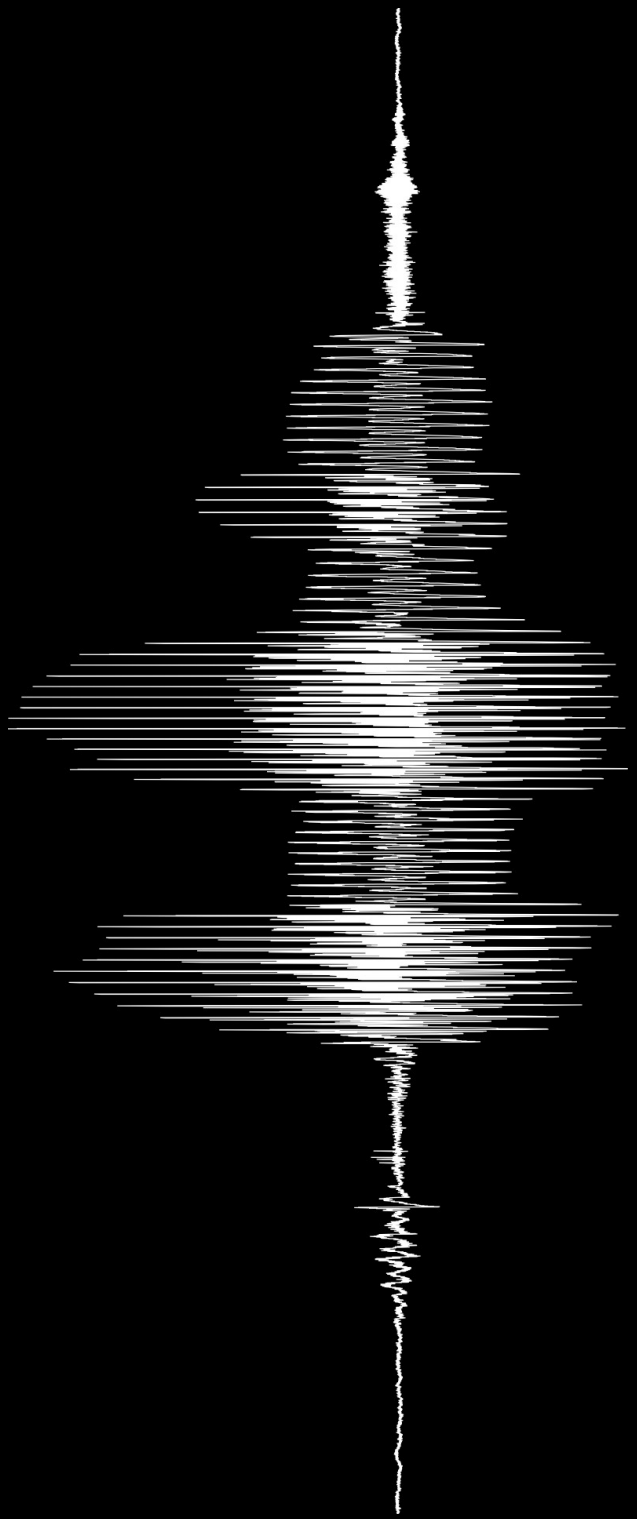
```
=     <>
>     >=    <=    <
+      -     *     /
^    mod
```



praat's scripting language

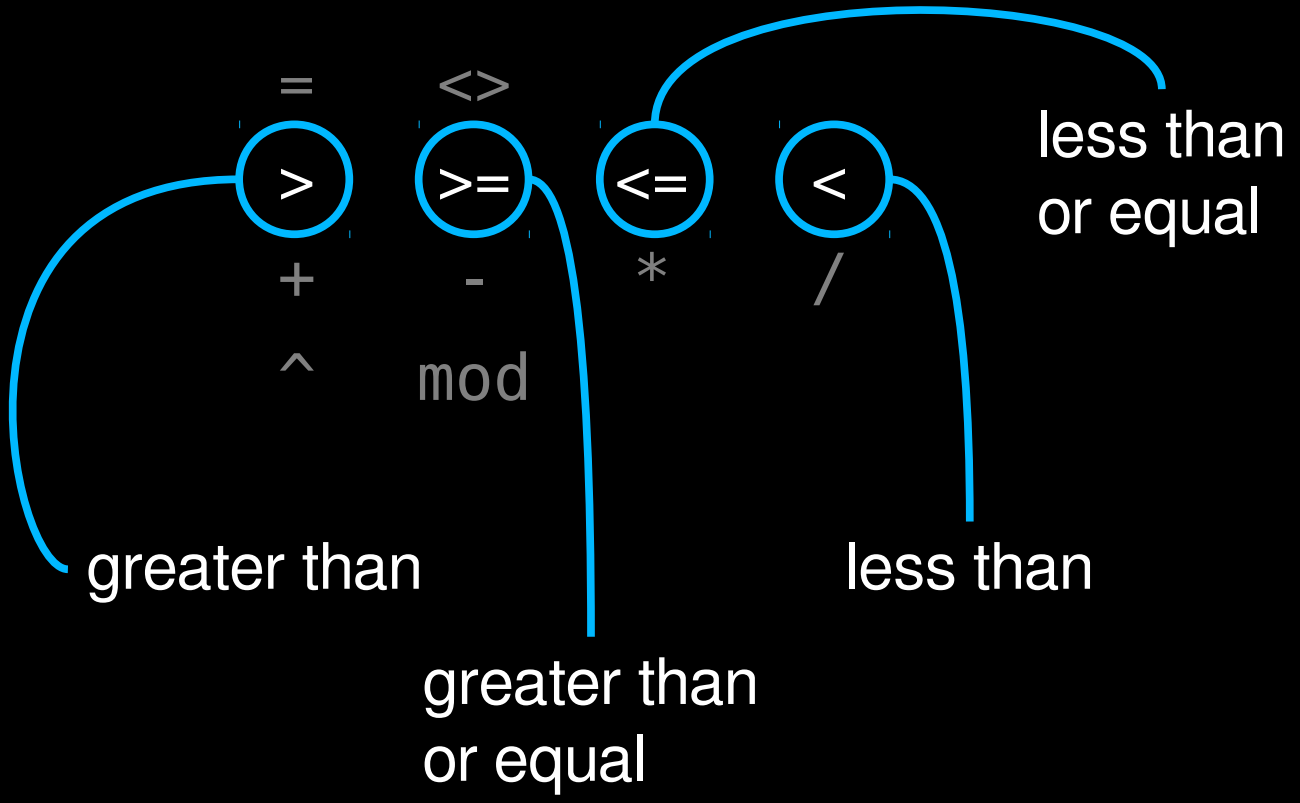
- operators

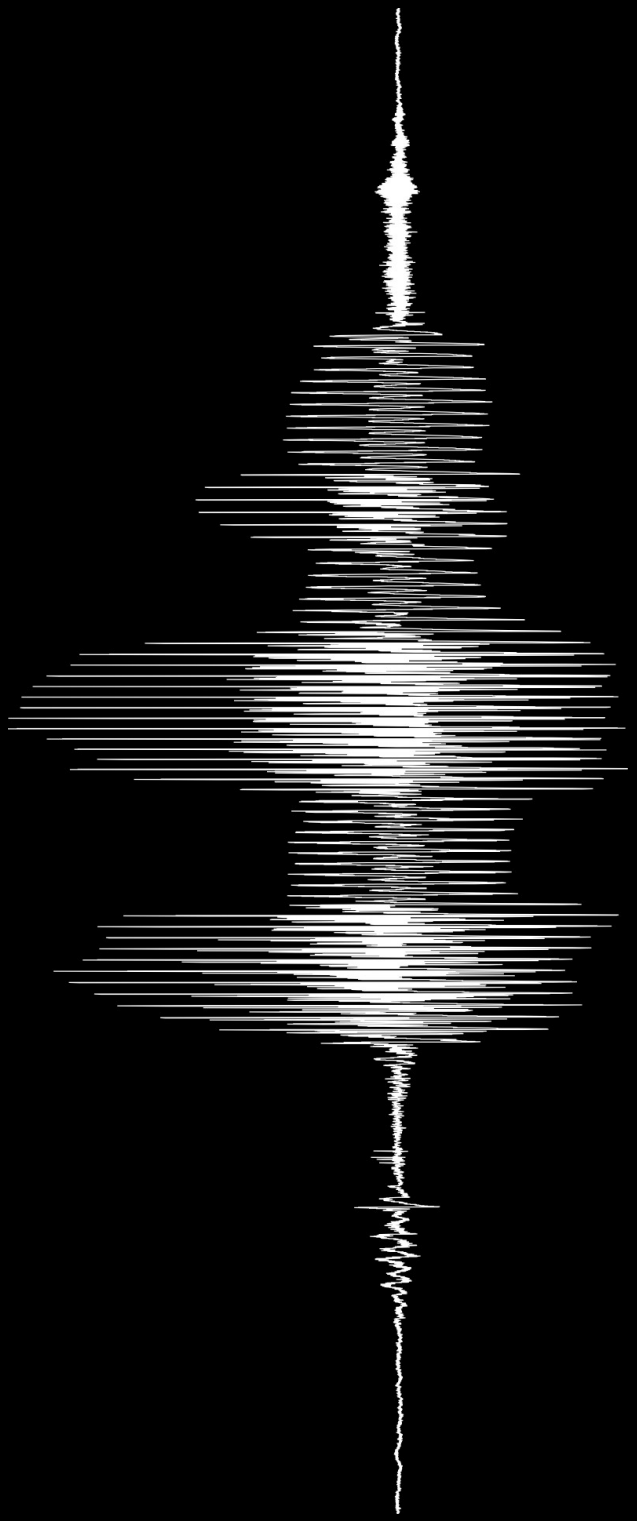




praat's scripting language

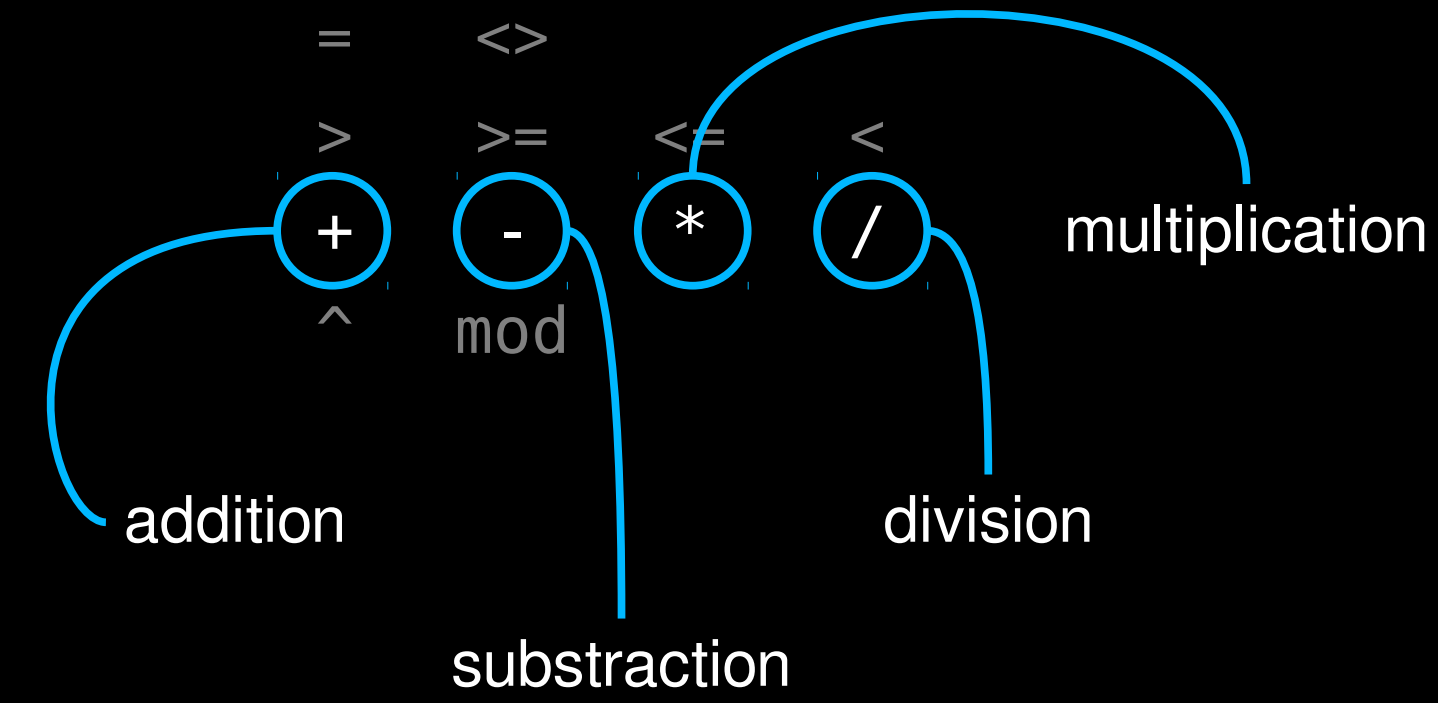
- operators

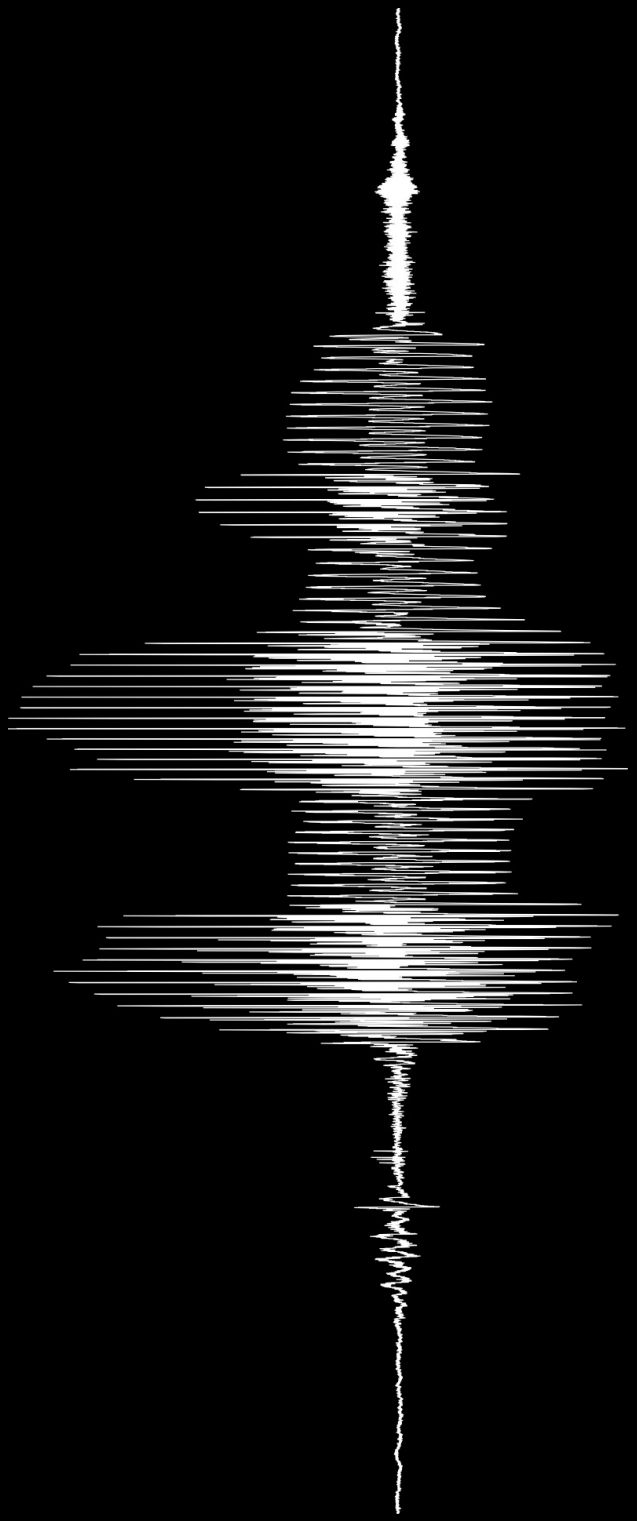




praat's scripting language

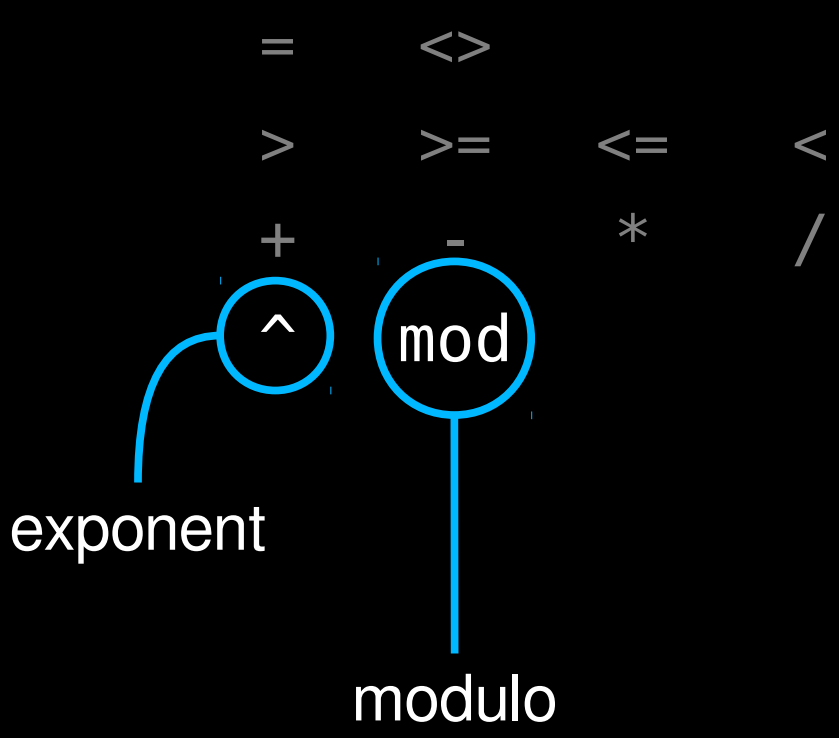
- operators

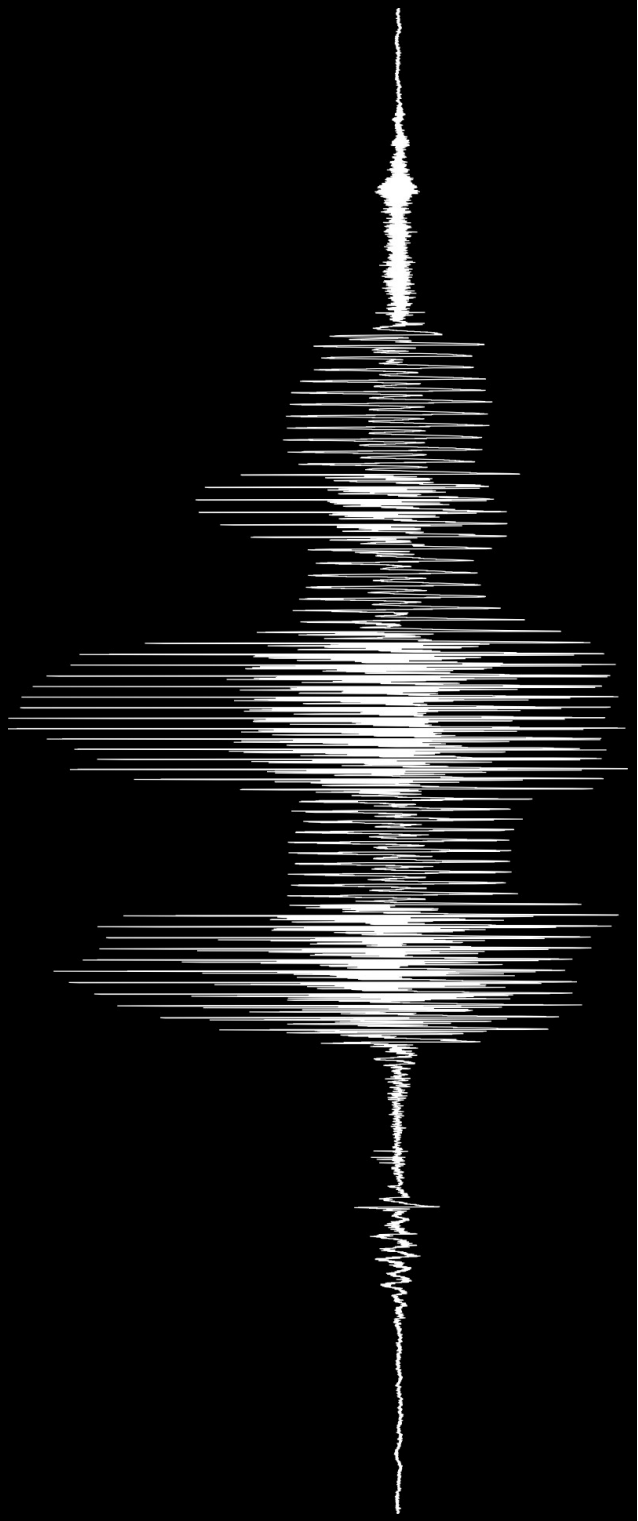




praat's scripting language

- operators





praat's scripting language

- `#comments`

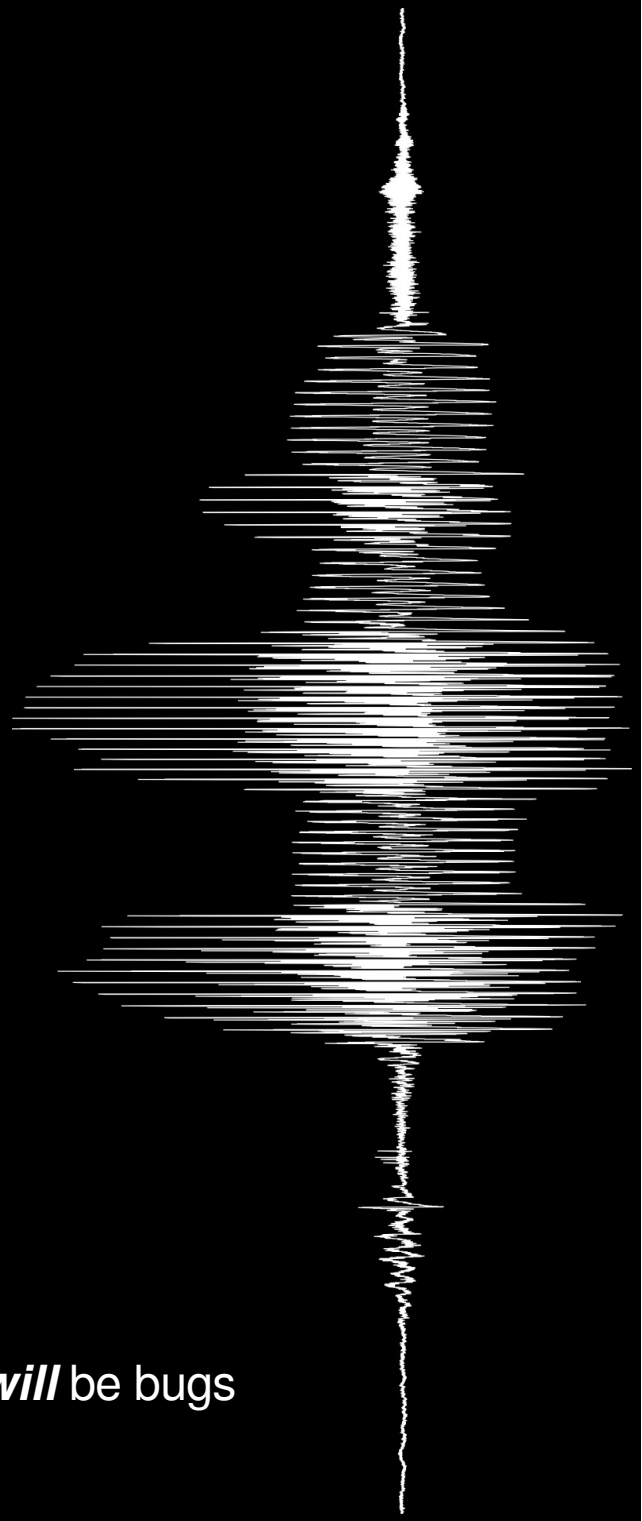
  are ignored by praat
- `clearinfo`

  clears the Info screen
- `printInfoLine(string$)`

  clears the Info screen and prints a line
- `appendInfoLine(string$)`

  adds a line to the Info screen

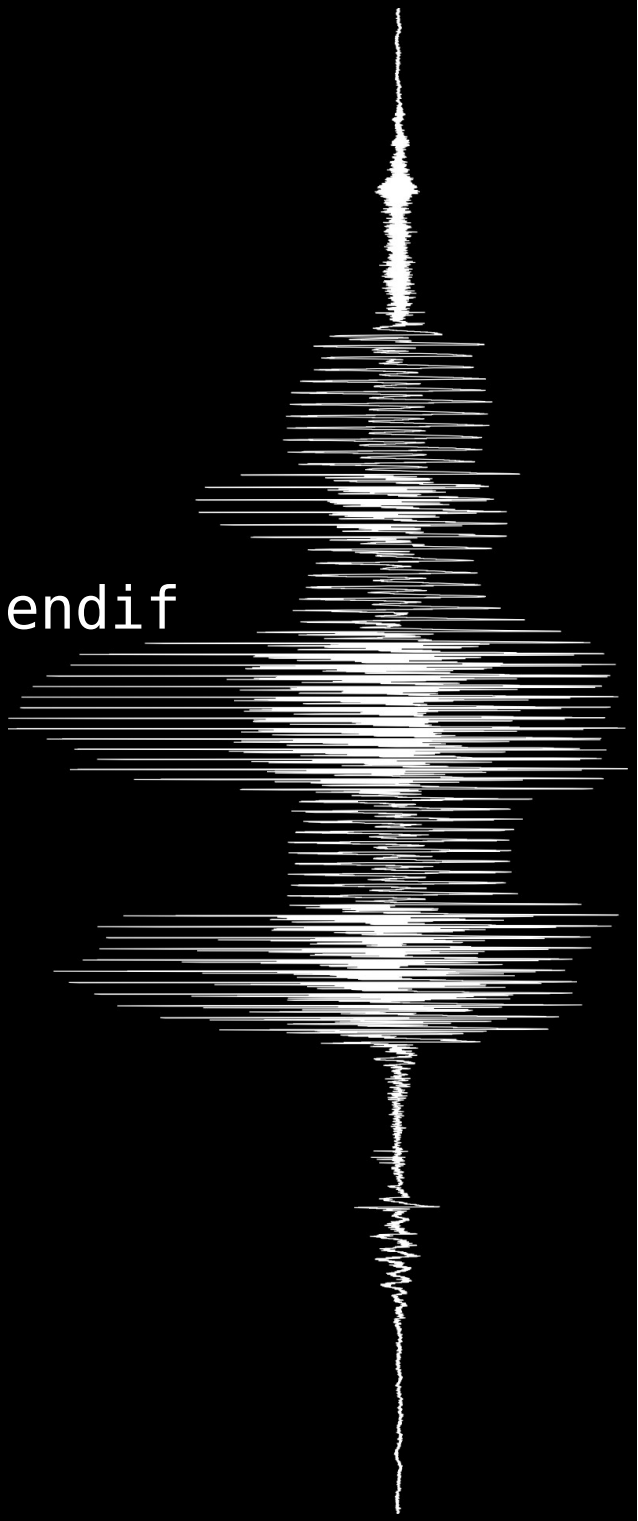
use `appendInfoLine()` for debugging code... for there ***will*** be bugs



praat's scripting language

- control structures
  - `for x [from y] to z … endfor`
  - `if cond1 … [elsif cond2 …] else … endif`
  - `while cond … endwhile`
  - `repeat … until cond`
- procedures

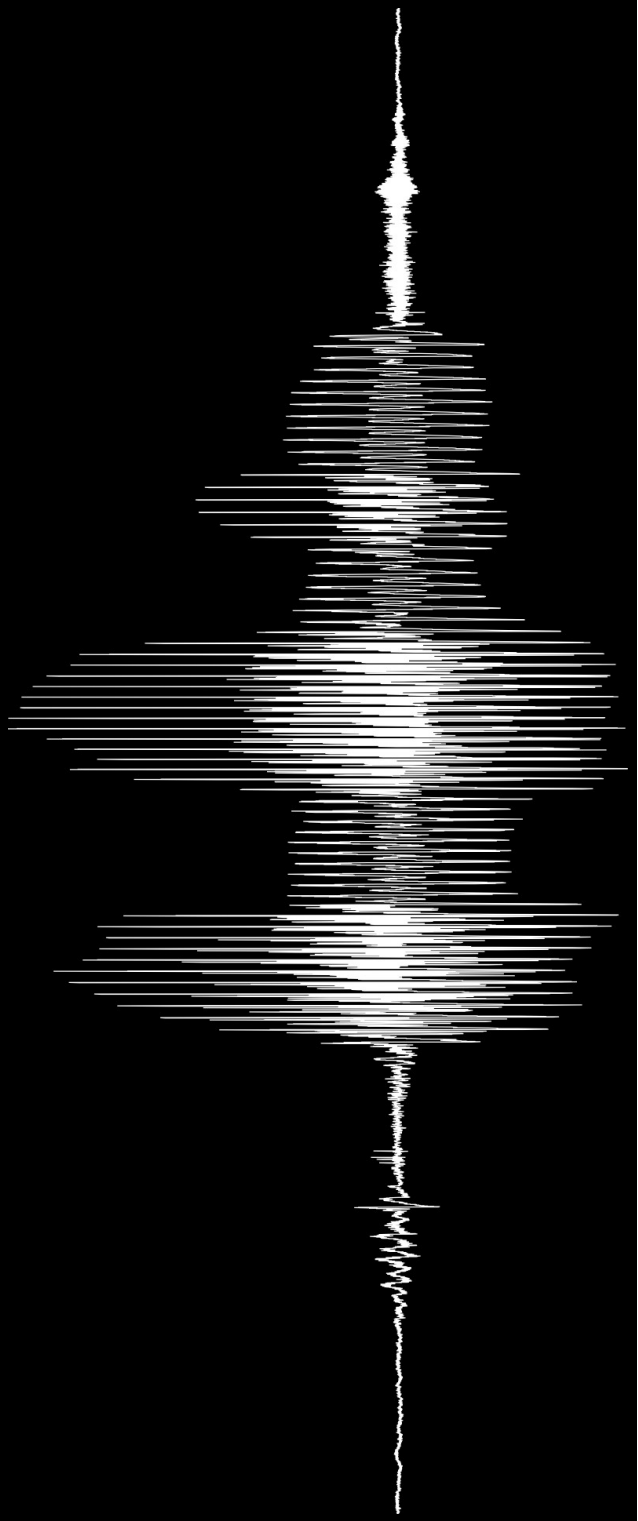
  (in more detail further along)



praat's scripting language

- control structures
  - `for x [from y] to z … endfor`

```
# example of for in praat

clearinfo
for number to 10
   count = 10 - number
   appendInfoLine(count, "...")
endfor
appendInfoLine("Liftoff!")
```

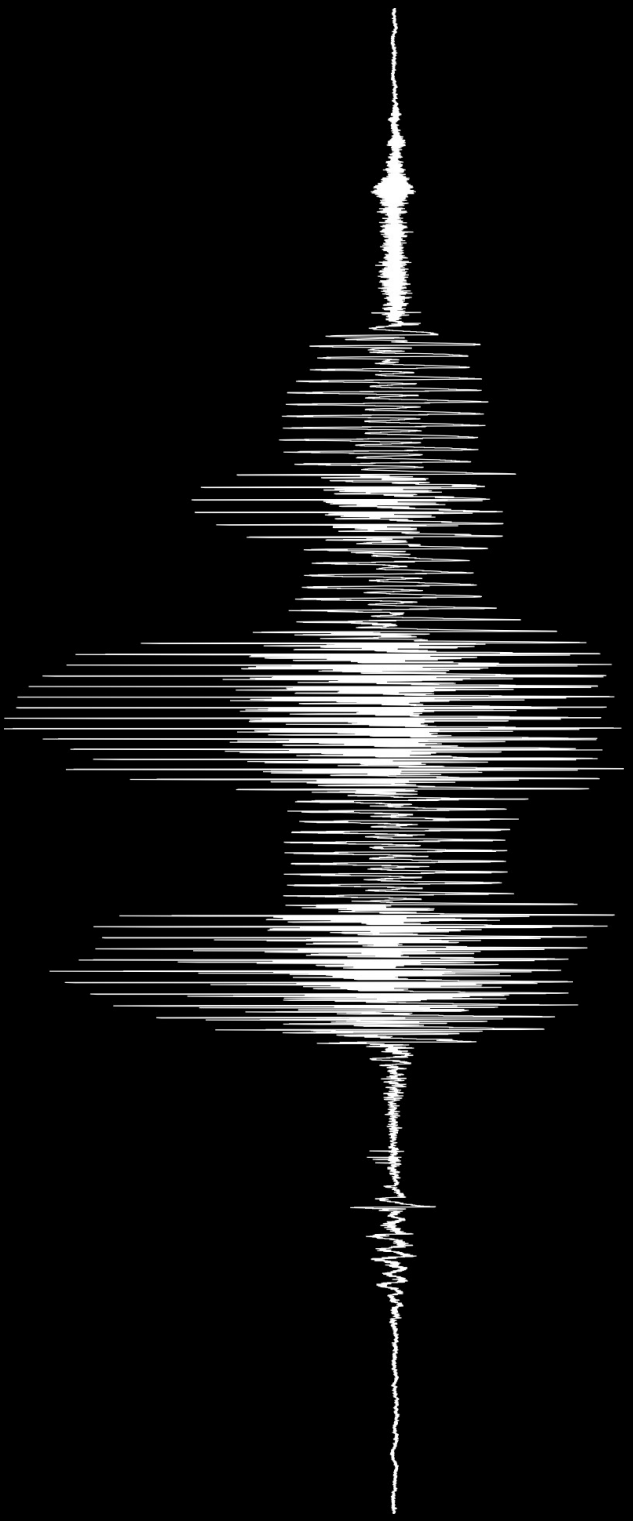


praat's scripting language

- control structures
  - `for x [from y] to z … endfor`

    in praat, a `for` block always increments the value of its control variable

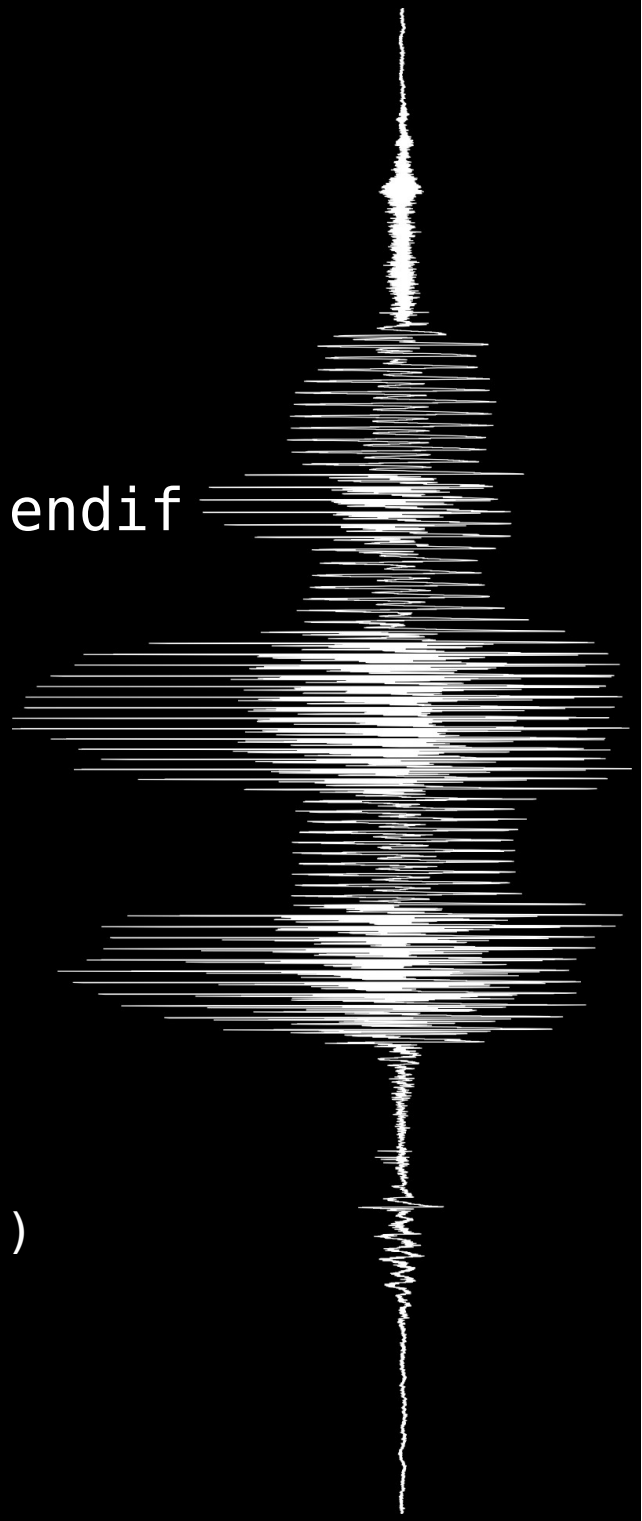
    if `y < z` the block is **never** executed



praat's scripting language

- control structures
  - `if cond1 … [elsif cond2 …] else … endif`

```
# example of if and for in praat

clearinfo
appendInfoLine("Start")
for number to 10
   if number < 5
      appendInfoLine("First half...")
   elsif number > 5
      appendInfoLine("Second half...")
   else
      appendInfoLine("Halfway through!")
   endif
endfor
appendInfoLine("And we are done!")
```



praat's scripting language

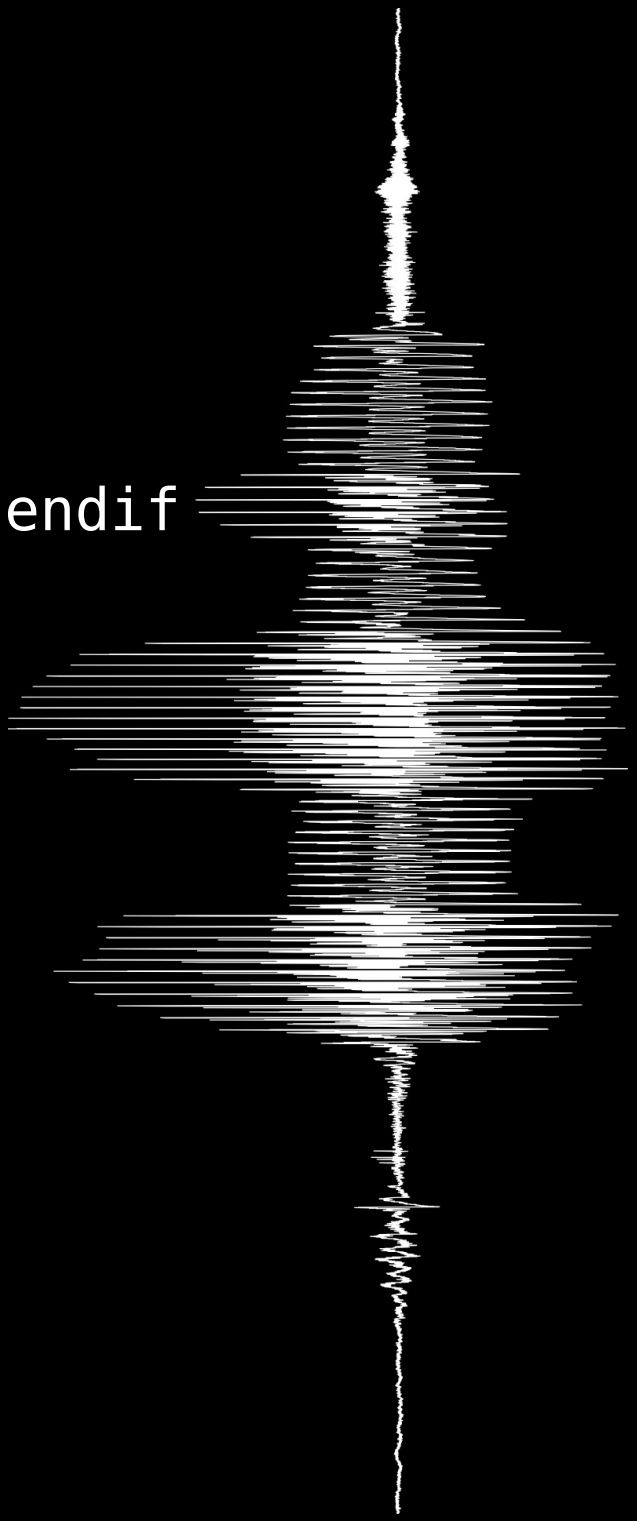
- control structures
  - `if cond1 … [elsif cond2 …] else … endif`

    any (defined) value that is not `0` or an empty string is true

    `elsif` can also be written `elif`

    `else` is very useful to define default values:

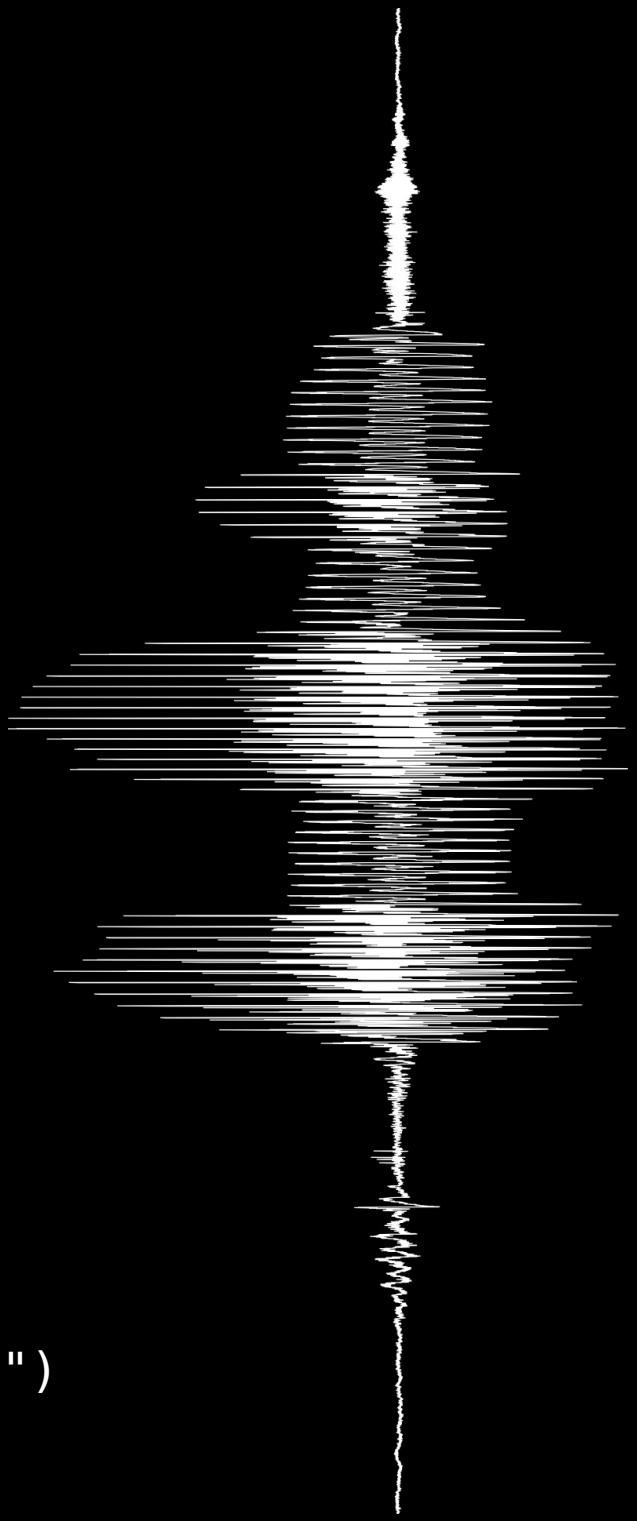
```
default_value = 0
f0 = do("Get value at time...", 0.5)
if f0 = undefined
  f0 = default_value
endif

# if there is no value, f0 will be 0
```



praat's scripting language

- control structures
  - repeat … until cond

```
# example of repeat in praat

clearinfo
number = 353467
appendInfoLine(
   ..."number starts as ", number)
repeat
   if number > 10
      number = number - number / 2
   elsif number < 10
      number = number + number / 2
   endif
   tmp = round(number)
   appendInfoLine(
      ..."...and now it is ", tmp, "...")
until round(number) = 10
appendInfoLine("And we are done!")
```



praat's scripting language

- control structures
  - `repeat … until cond`

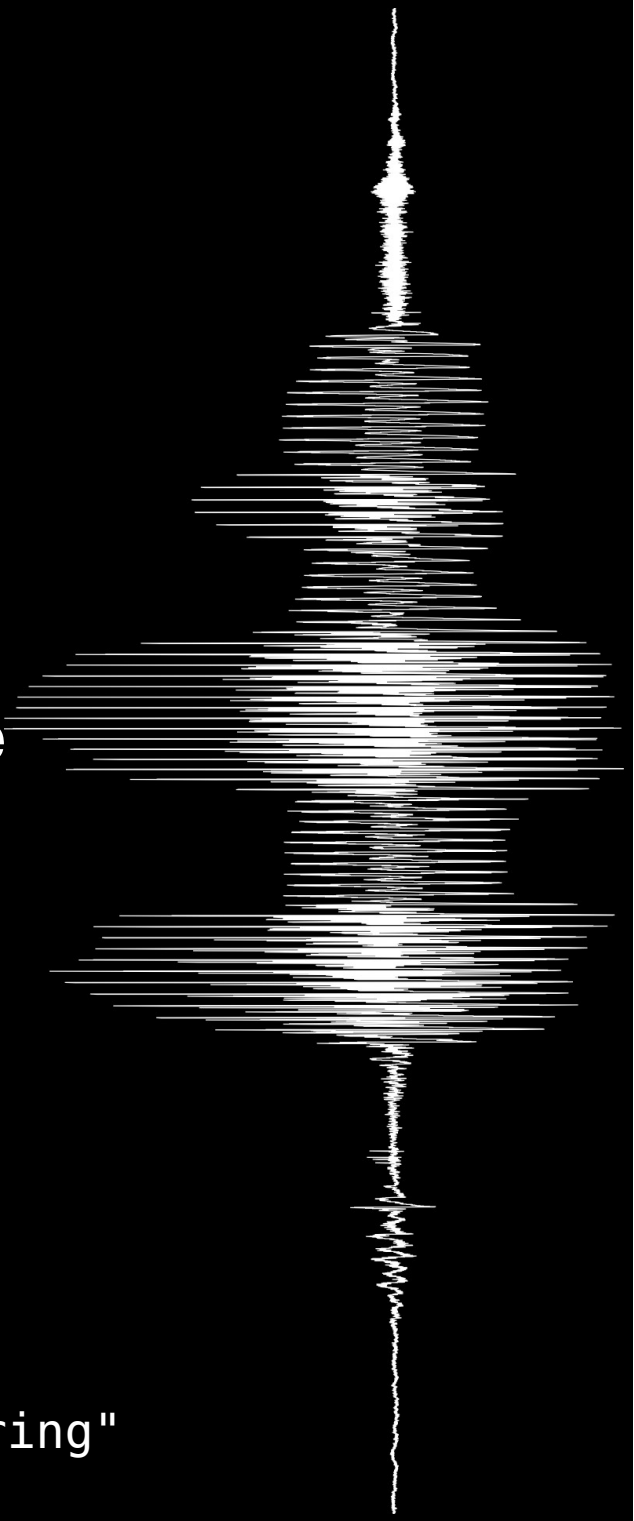
    the block executes until the condition is true

    it is possible to create infinite loops
    (use with care!)

    the condition is tested at the ***end*** of each loop

    long lines can be broken with ellipsis at the beginning of the next line:

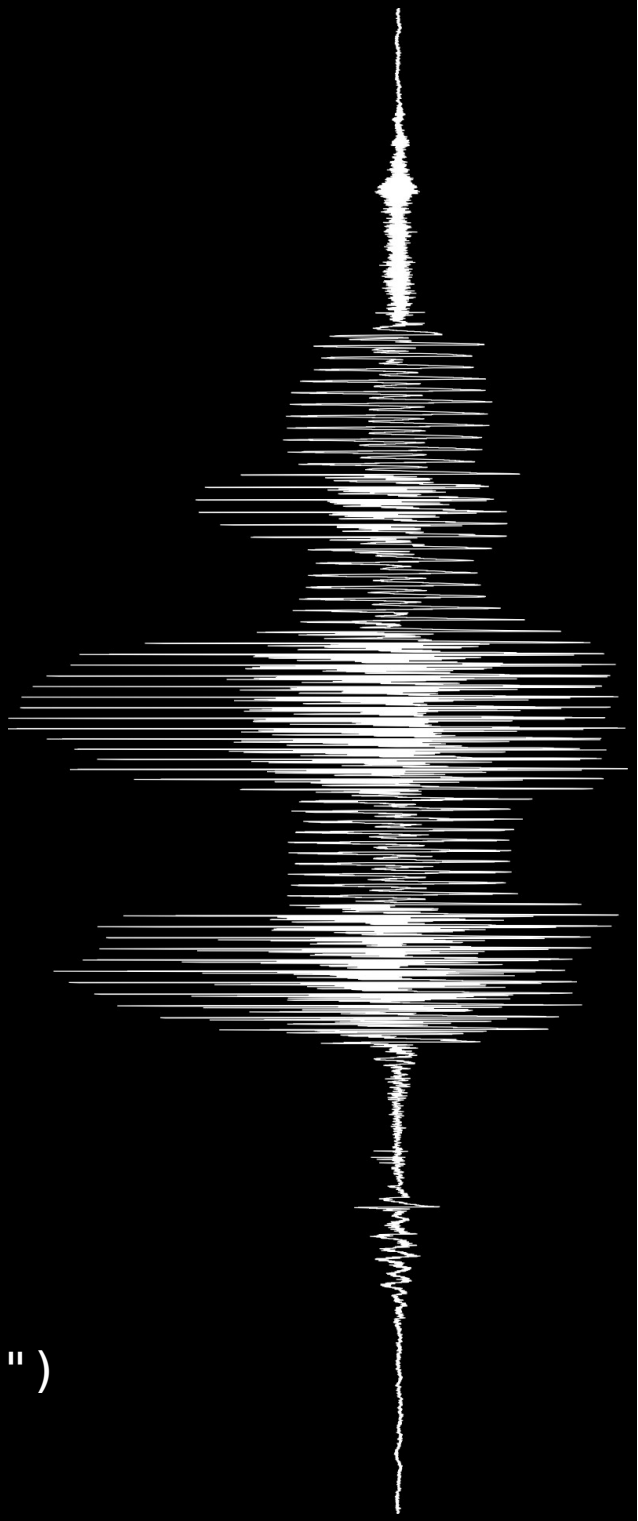
```
string$ = "even though this is a very
...long string, it is still just one string"
```



praat's scripting language

- control structures
  - while cond … endwhile

```
# example of while in praat

clearinfo
number = 353467
appendInfoLine(
   ..."number starts as ", number)
while round(number) <> 10
   if number > 10
      number = number - number / 2
   elsif number < 10
      number = number + number / 2
   endif
   tmp = round(number)
   appendInfoLine(
      ..."...and now it is ", tmp, "...")
endwhile
appendInfoLine("And we are done!")
```



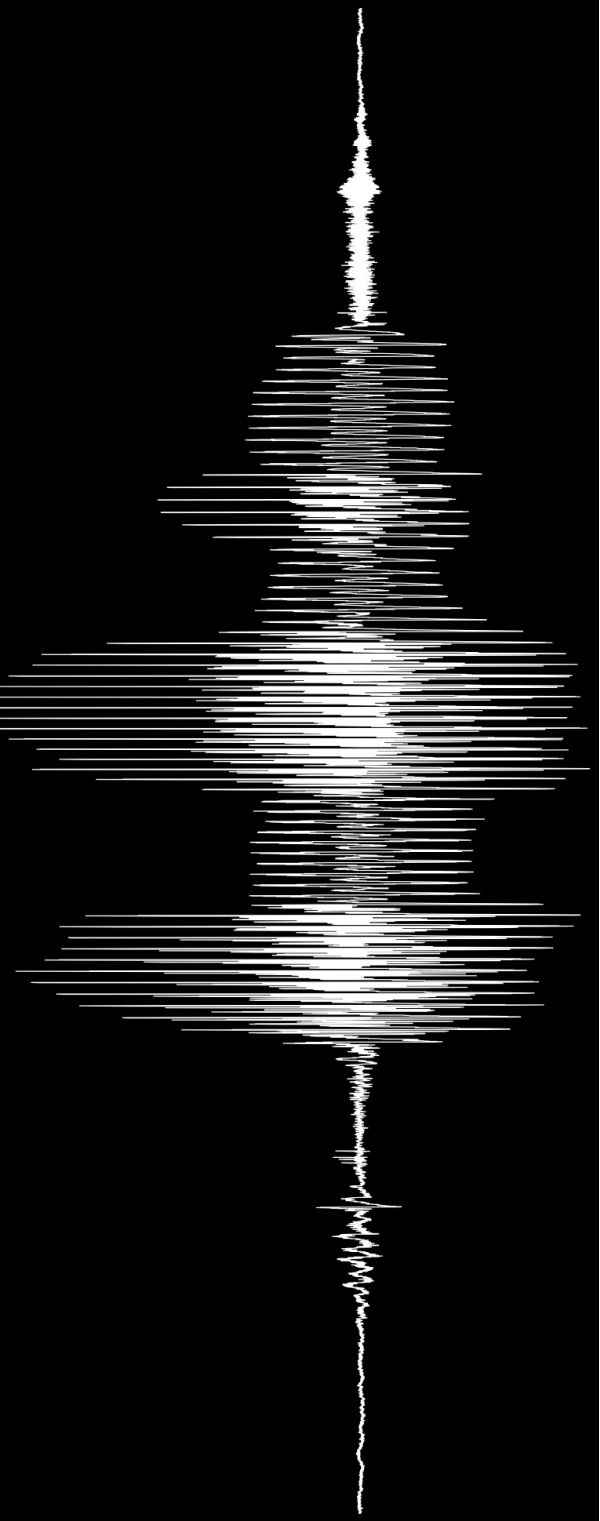
praat's scripting language

- control structures
  - `while cond … endwhile`

    the block executes while the condition is true

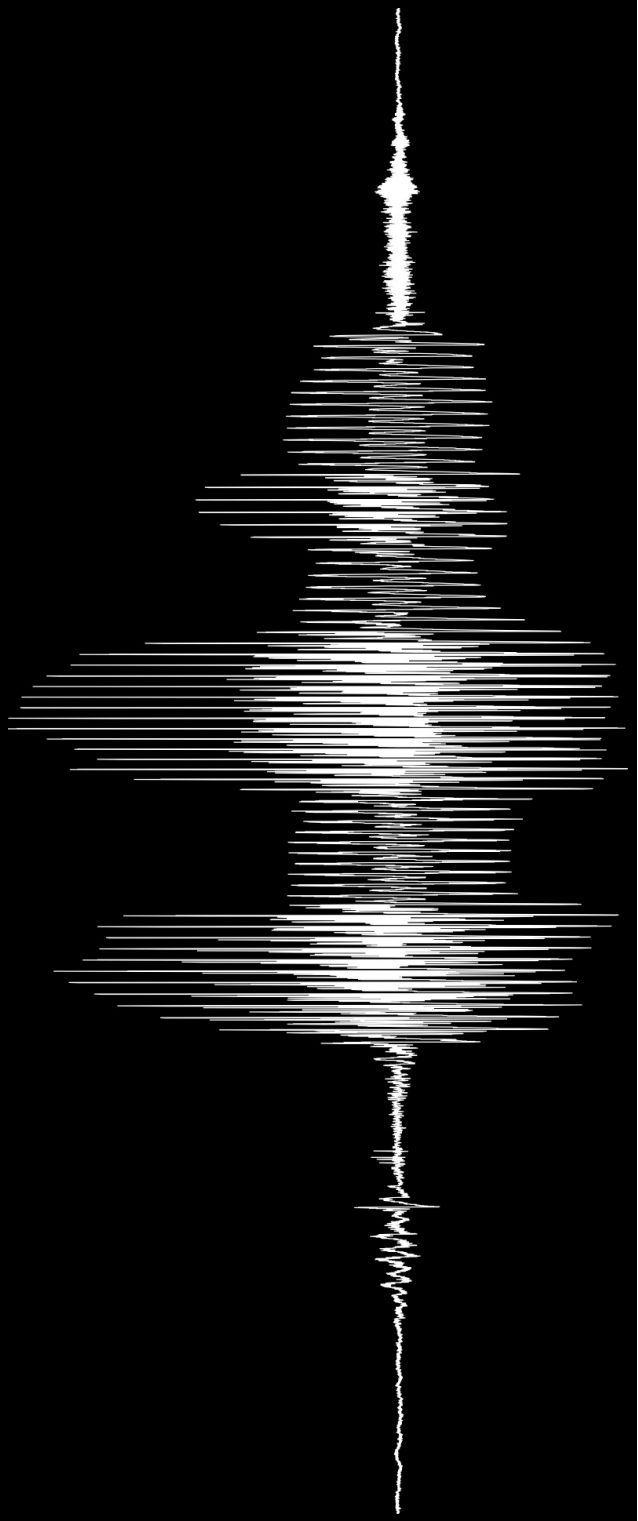
    it is possible to create infinite loops
    (use with care!)

    the condition is tested at the ***beginning*** of each loop



praat's scripting language

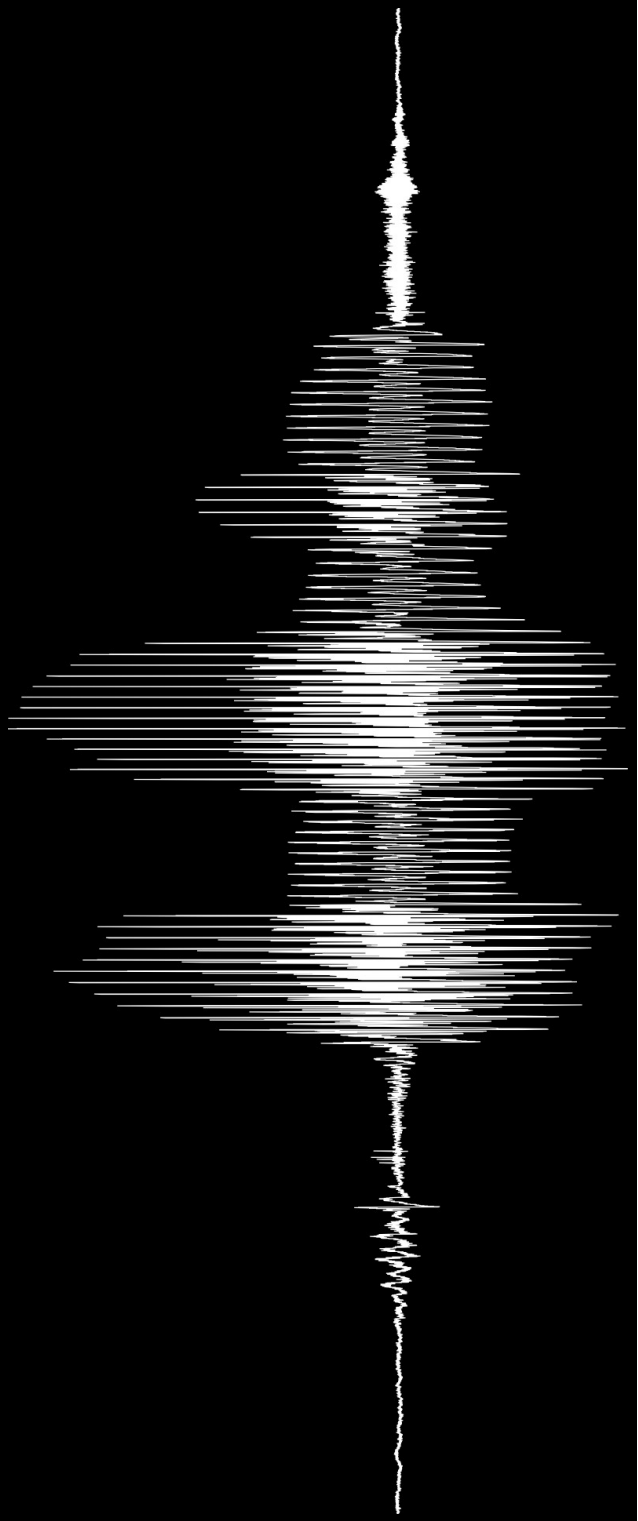
- logical operators
  - `and (&)`
  - `or (|)`
  - `not (!)`
- functions*
  - string functions
  - numeric functions



* more on this on part 5!

intermission

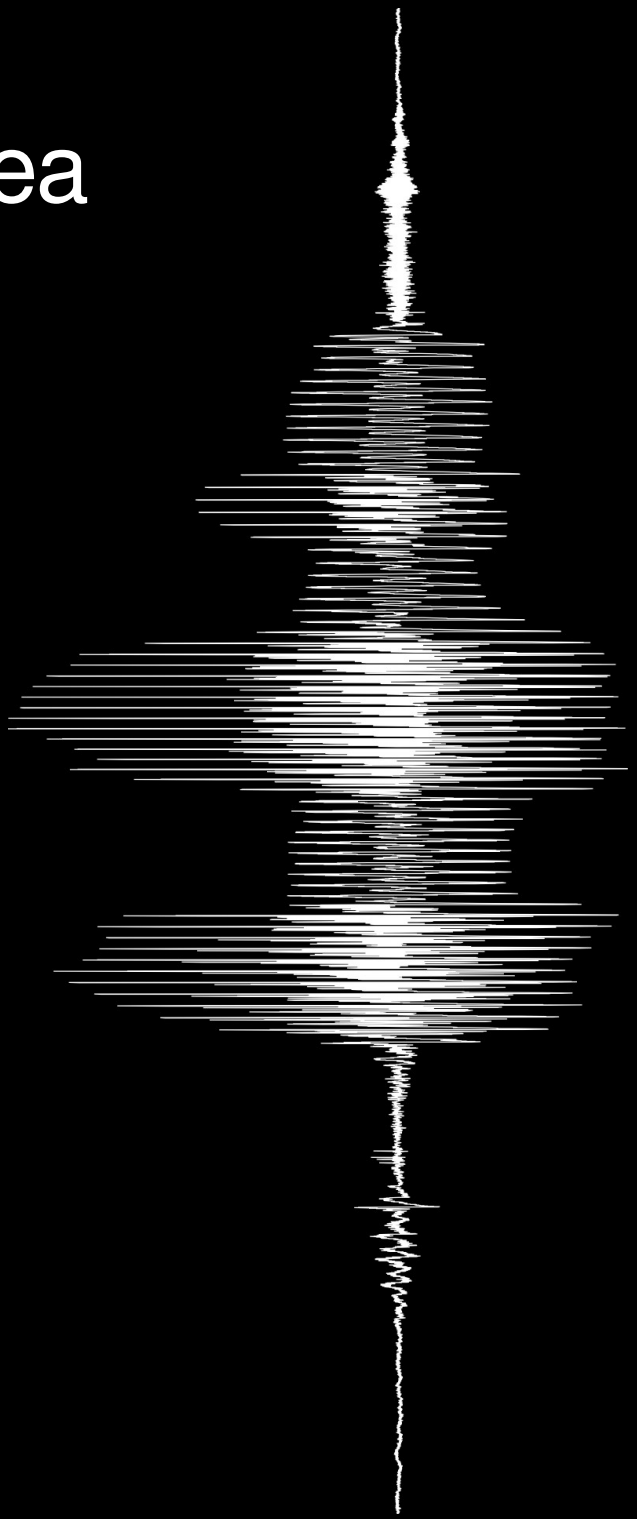
how about those good practices?



bad idea

```
top = 100
for q from 3 to top
is = 1
p = q mod 2
if p = 0
is = 0
endif
this = 3
while this <= sqrt(q)
p = q mod this
if p = 0
is = 0
endif
this = this + 2
endwhile
if is = 1
printline 'q'
endif
endfor
```

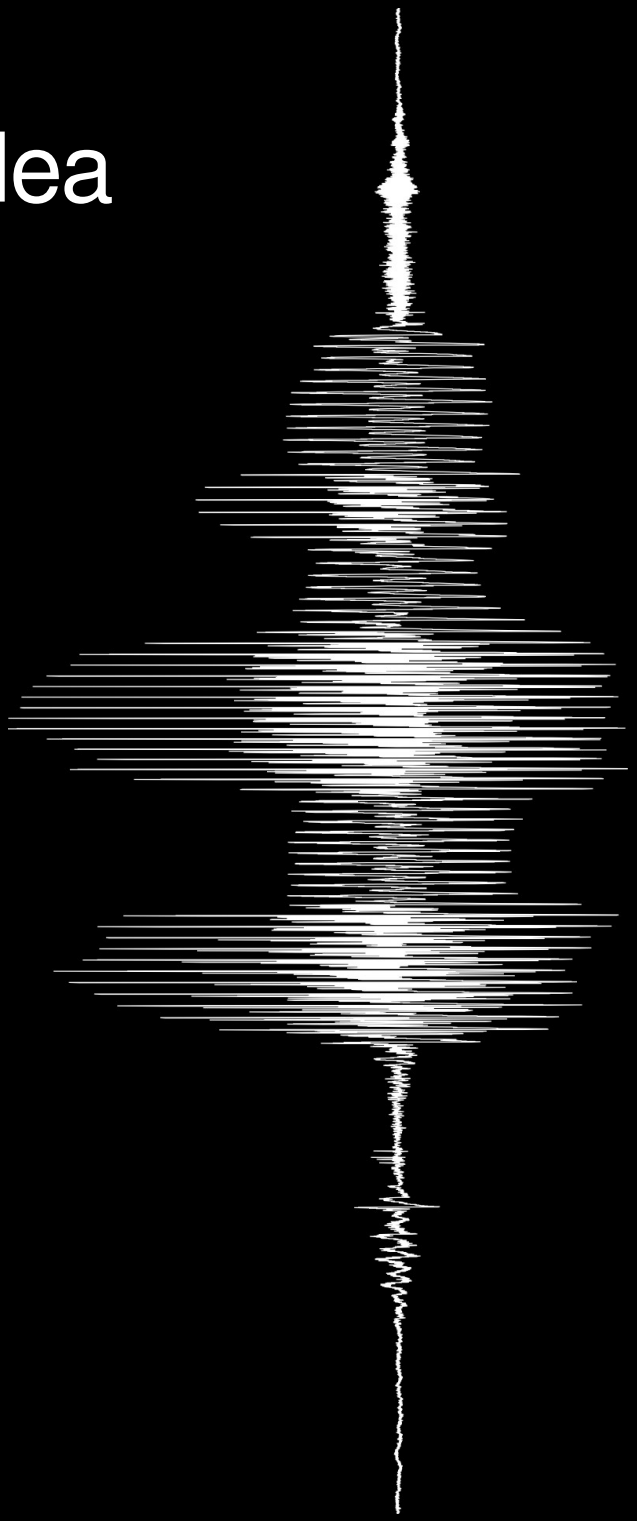
- ~~easy to read~~
- ~~clear~~
- ~~extensible~~
- it works
- robust?
- efficient?



good idea

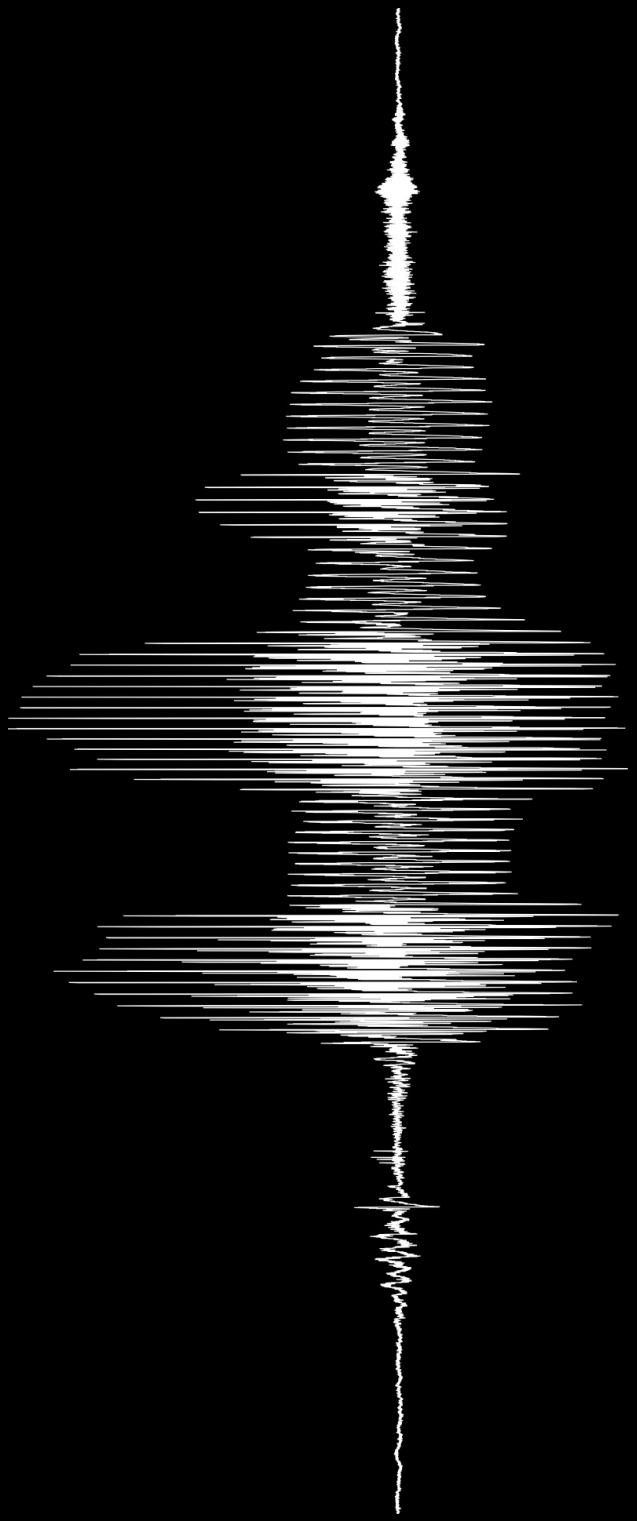
```
# detect prime numbers by brute force
# start from 3 to skip even numbers

clearinfo

# for every number up to limit
limit = 100
for n from 3 to limit
    prime = 1
    for candidate from 2 to (n-1)
        test = n mod candidate
        if test = 0
            prime = 0
        endif
    endfor
    if prime
        printline 'n'
    endif
endfor
```

- easy to read
- clear
- ~~extensible~~
- it works
- robust
- ~~efficient~~

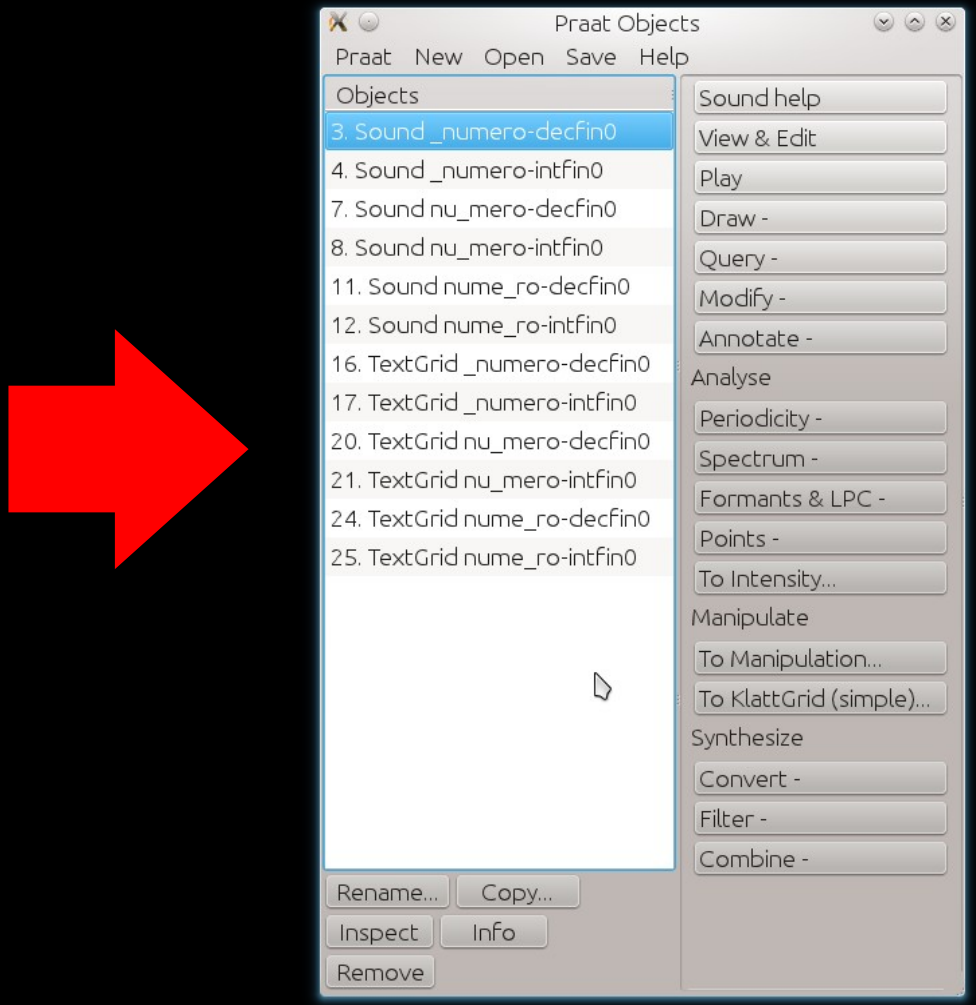


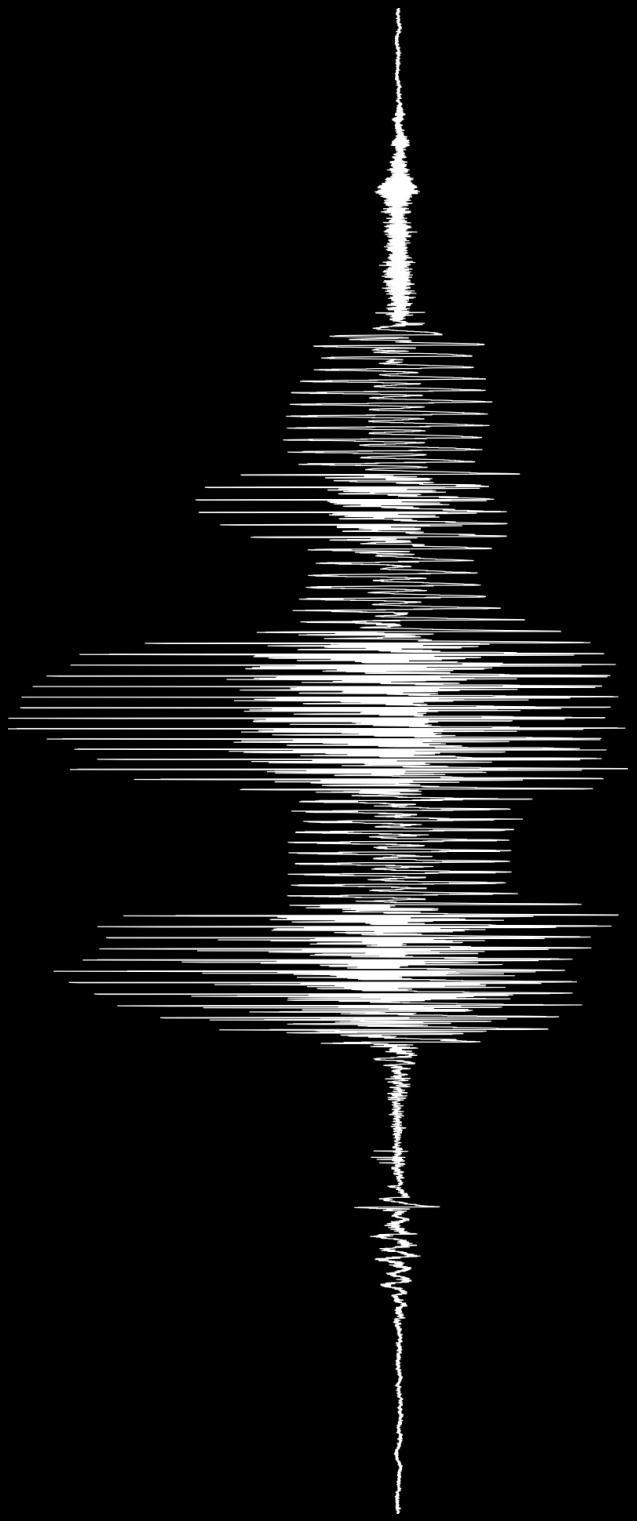
part 4

object handling



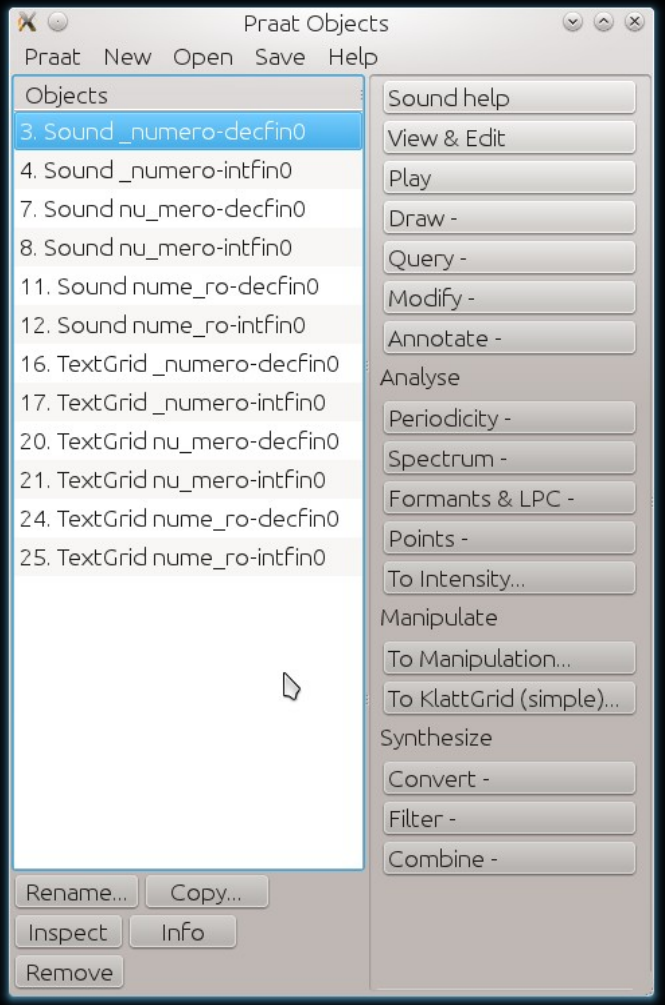
the object window





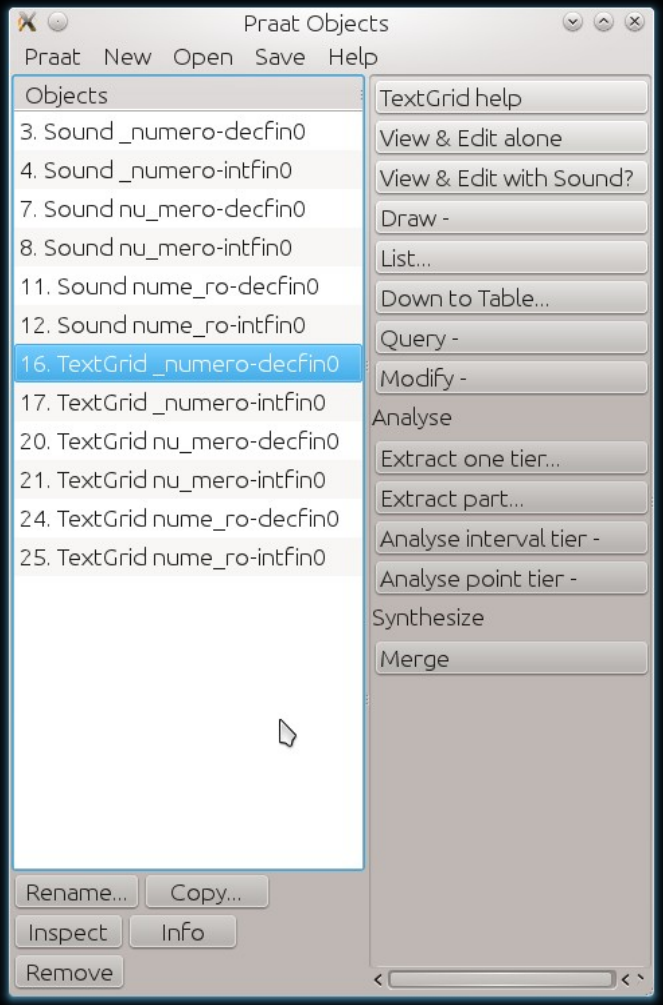
on the left we have the list of objects...

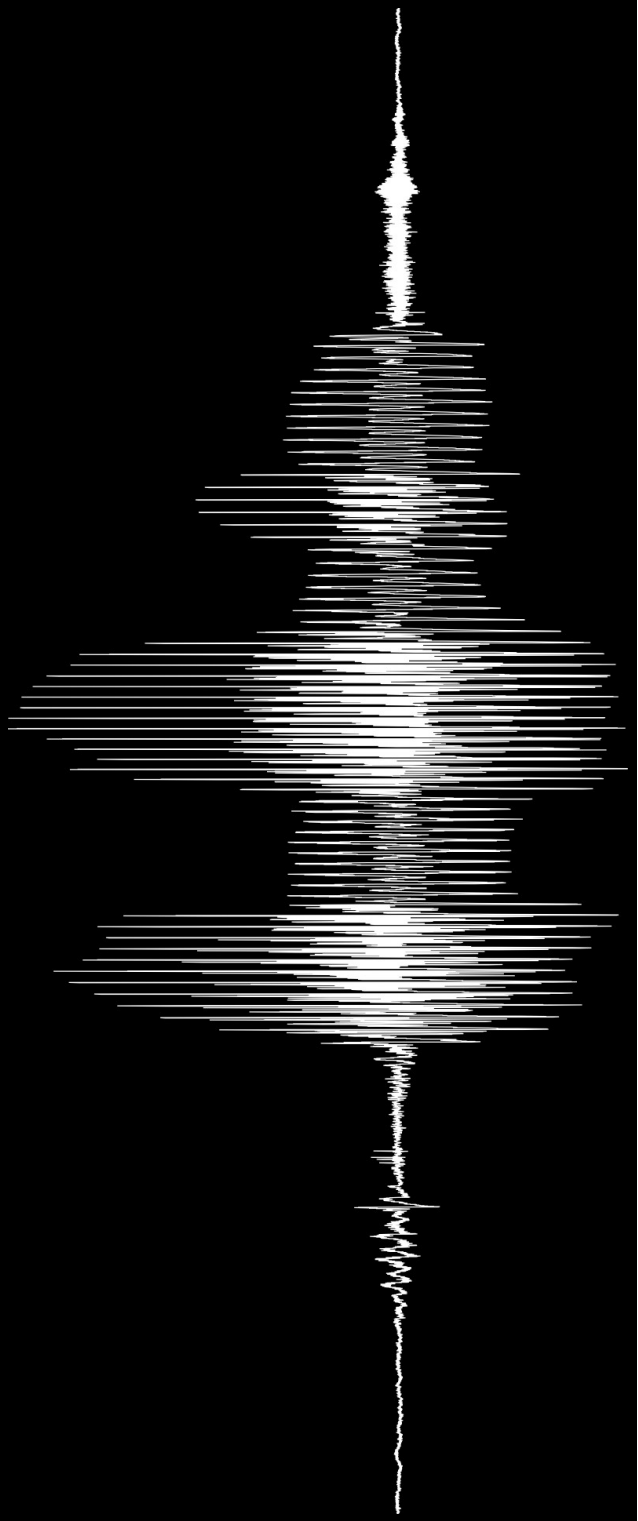
the object window



...and on the right the available actions

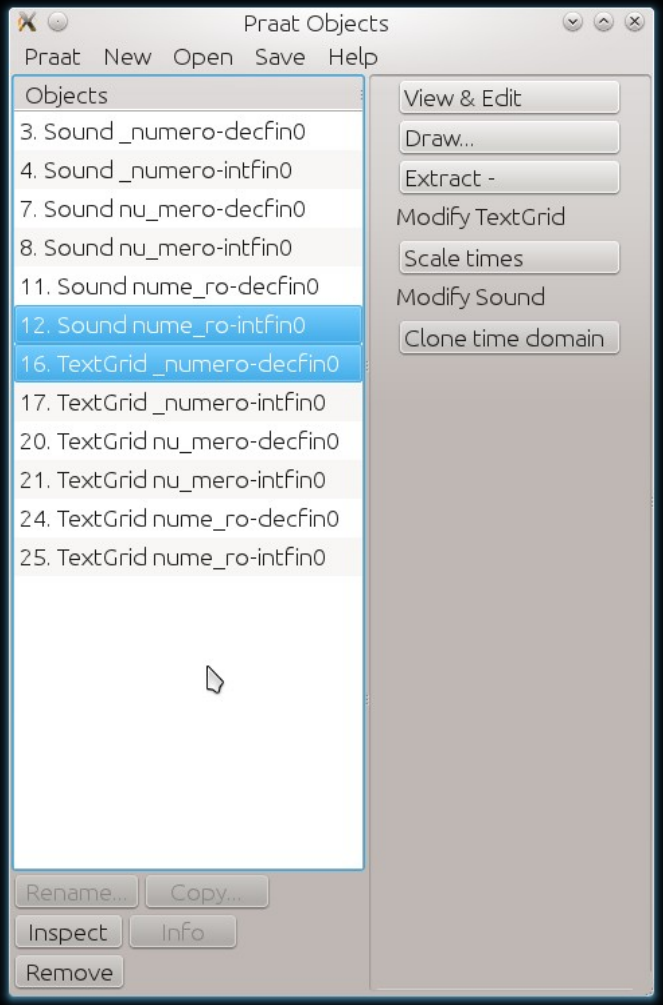
the object window



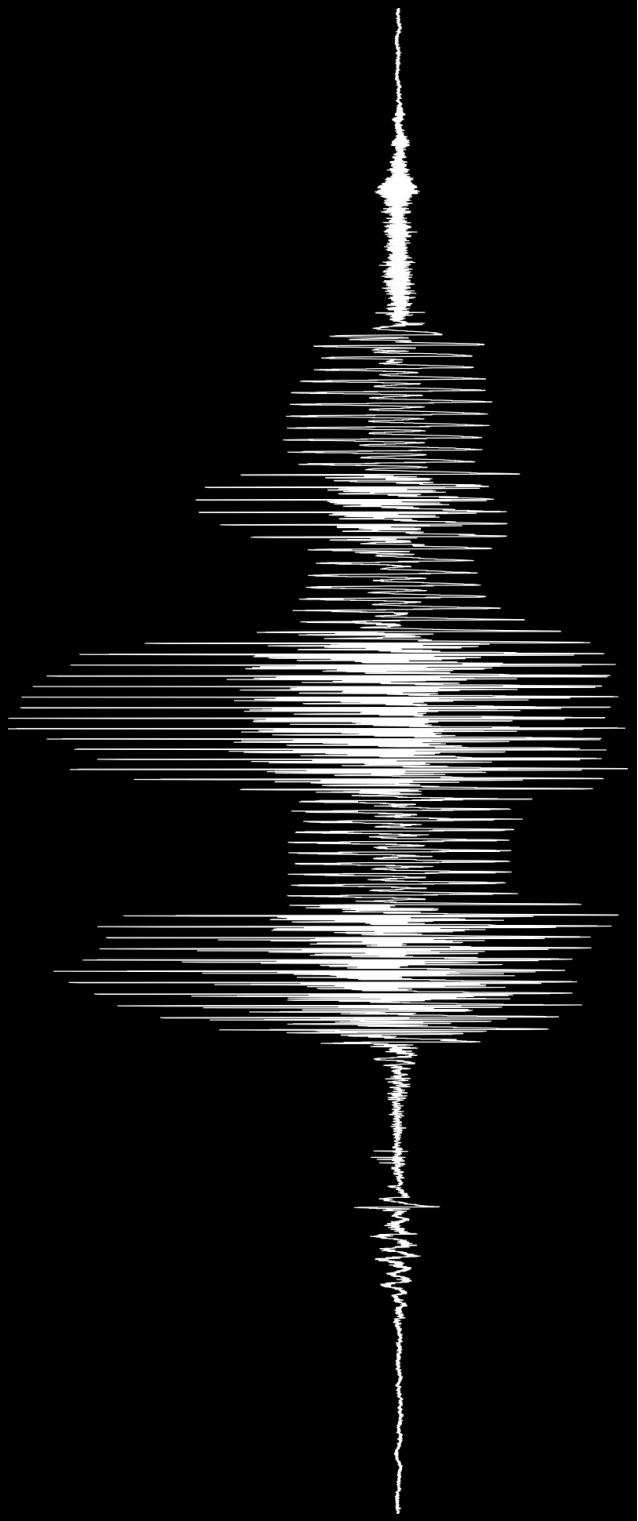


but these are selection-sensitive

the object window

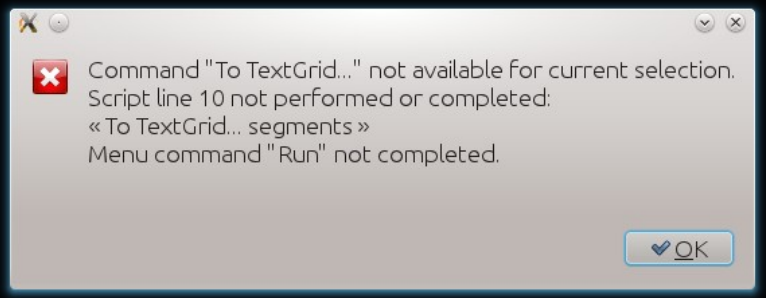


and some actions are only available
for combinations of objects



and why should we care?

- because our active selection determines our available options
- and those options are the same options we'll have available from within a script



object selection

praat has some functions to manipulate the active selection:

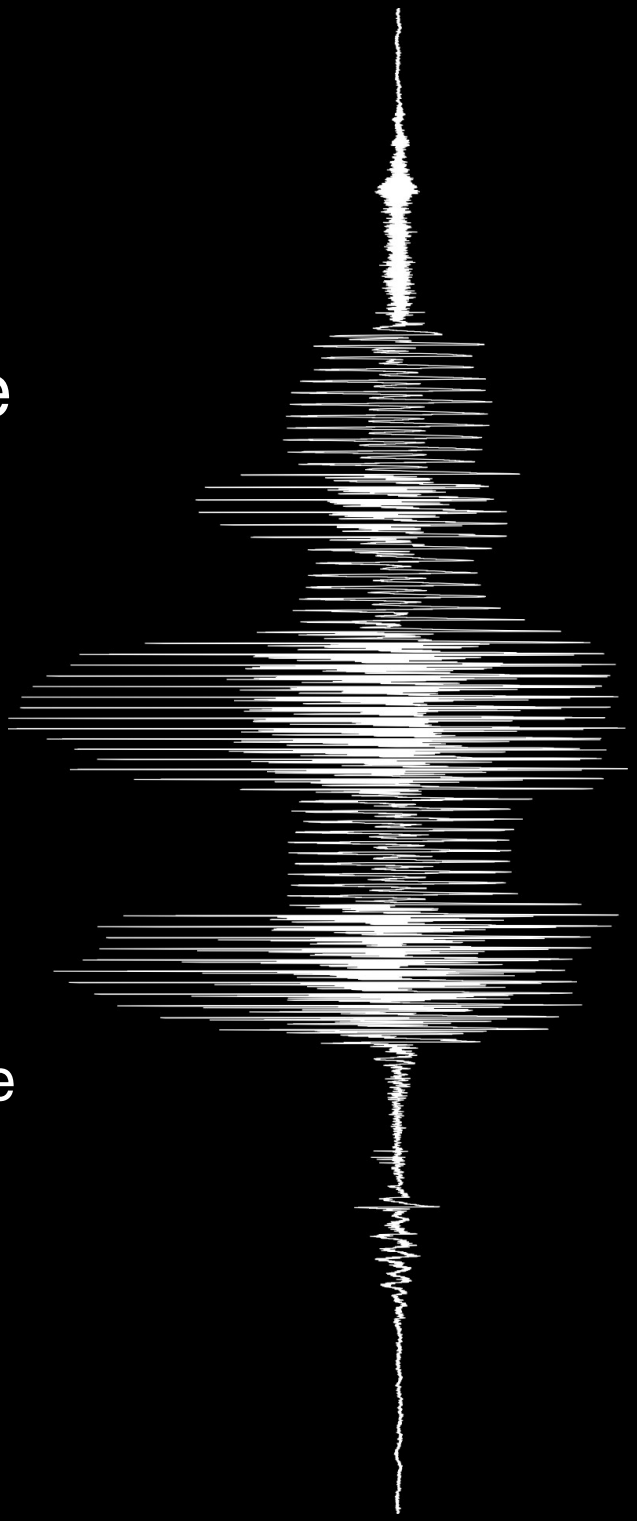
- `selectObject(obj1[,obj2,…])`

  selects the specified objects

- `plusObject(obj1[,obj2,…])`

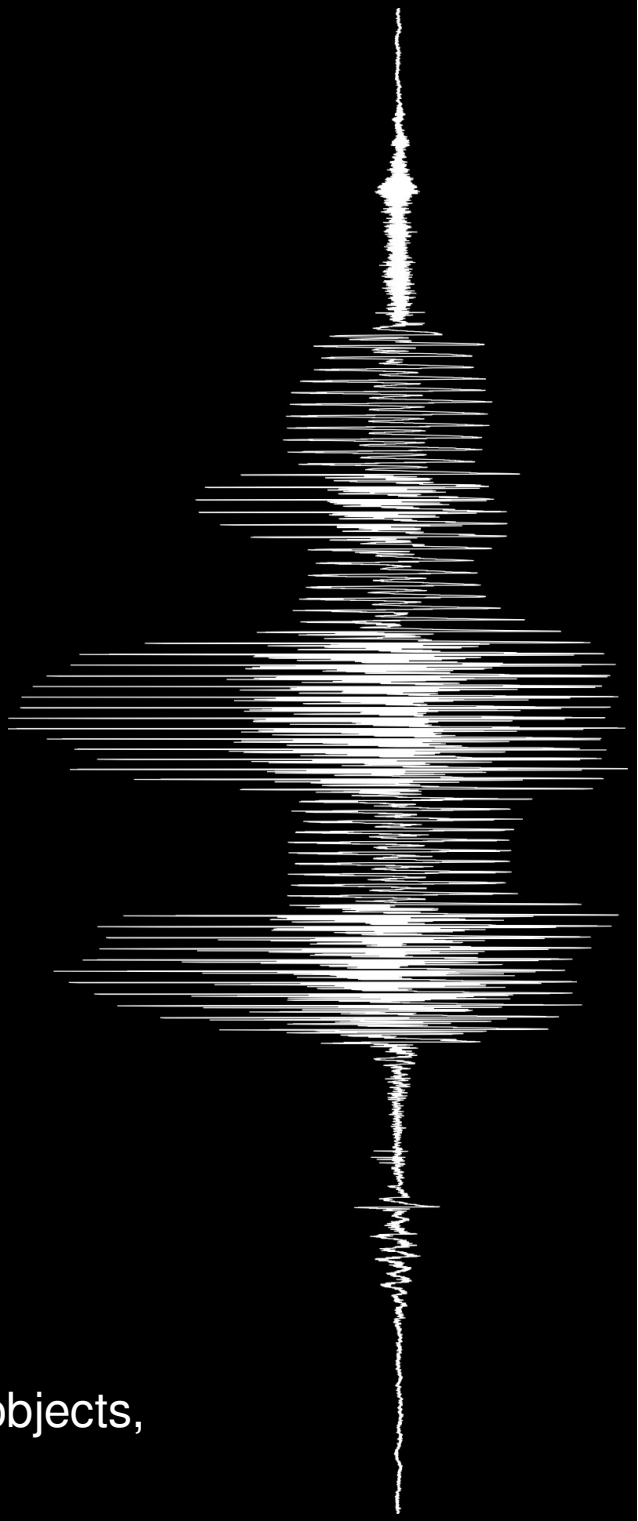
  `minusObject(obj1[,obj2,…])`

  adds or removes the specified objects to the current selection



object selection

- scripts inherit the selection that was active when they were run
- this selection is modified whenever you:
  - actively change selection
  - remove selected object(s)
  - create a new object
    (new objects are automatically selected)



if you'll be working with lots of objects,
save that selection!

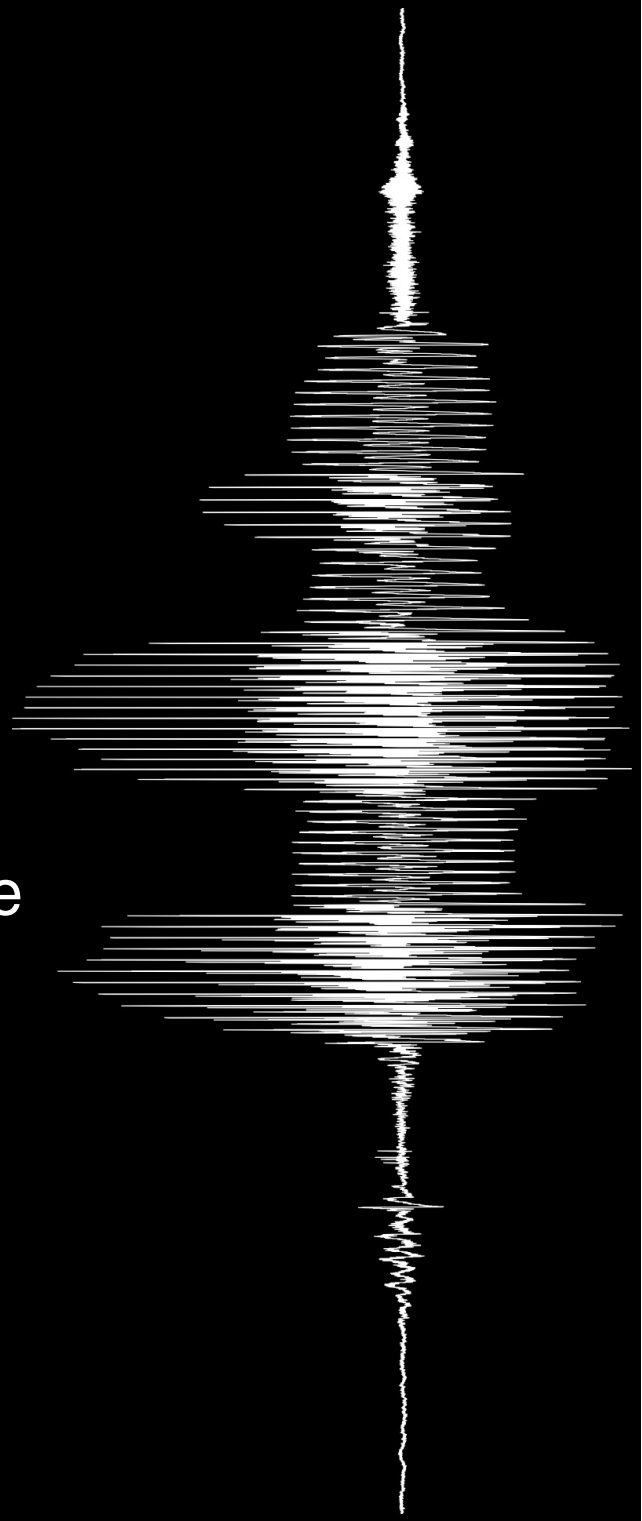
object selection

- two (or more) objects may have the same name
- use instead object IDs which are unique

```
selectObject(3,4,7,8)
minusObject(4,8)
```

since object IDs are assigned sequentially, they may skip some numbers at times

object selection



what is currently selected?

- `numberOfSelected(type$)`

  returns the number of selected objects of type `type$`**
- `selected(type$, #)`

  returns the ID of #th* selected object of type `type$`**
- `selected$(type$, #)`

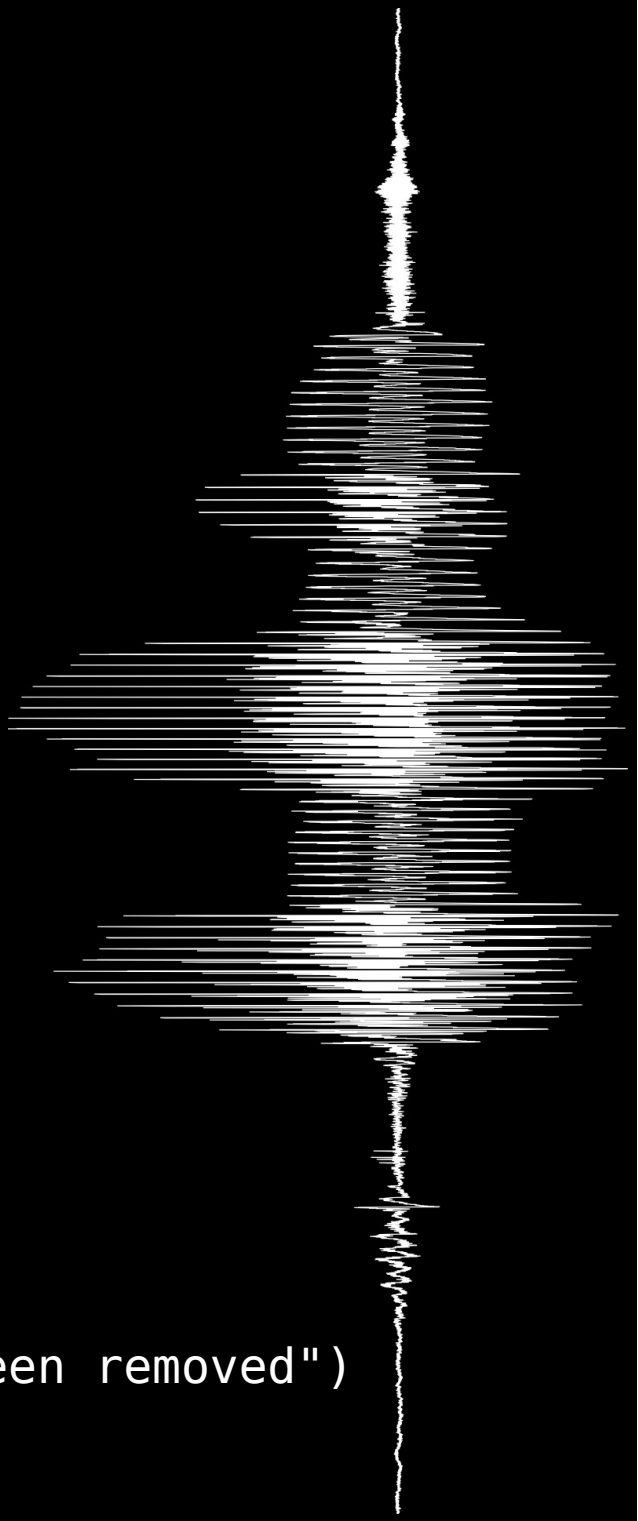
  returns the name of #th* selected object of type `type$`**

* if not provided, this will be 1 by default

** if not provided, objects of all types will be considered

how this all fits together

```
# example of selection of objects by type
# number of selected "Sound" objects

n = numberOfSelected("Sound")

# save original selection

for i to n
    object[i] = selected("Sound", i)
endfor

# for each originally selected object
for i to n
    selectObject(object[i])
    # get its name
    name$ = selected$()
    # and remove it
    removeObject(object[i])
    appendInfoLine("Object ", name$, " has been removed")
endfor
```



the push of a button

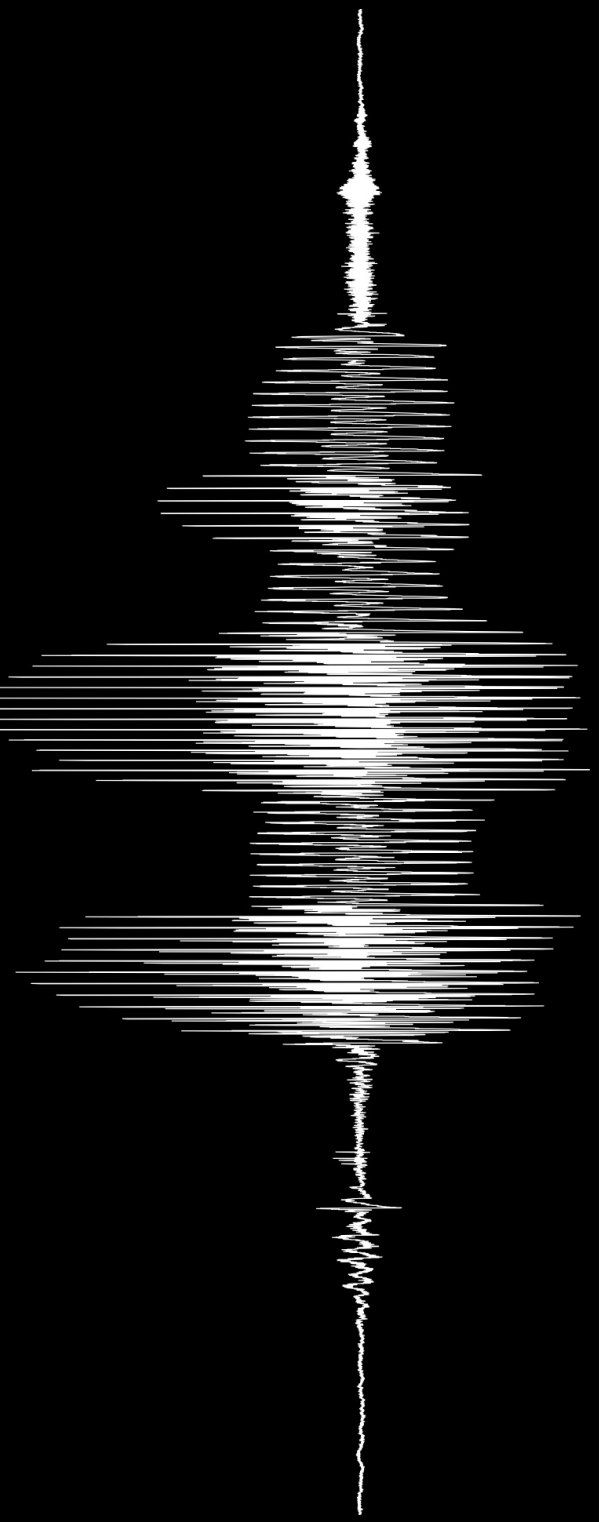
the buttons we can normally use with our mouse we can also use within scripts, and they work in the same way:

- if it needs arguments we'll provide them*
- if it returns some information it will do so

    and we can assign that to a variable!

- to use them we use the `do( )` and `do$( )` functions

    the latter for when it returns a string



* these have names that end with an ellipsis...

the push of a button

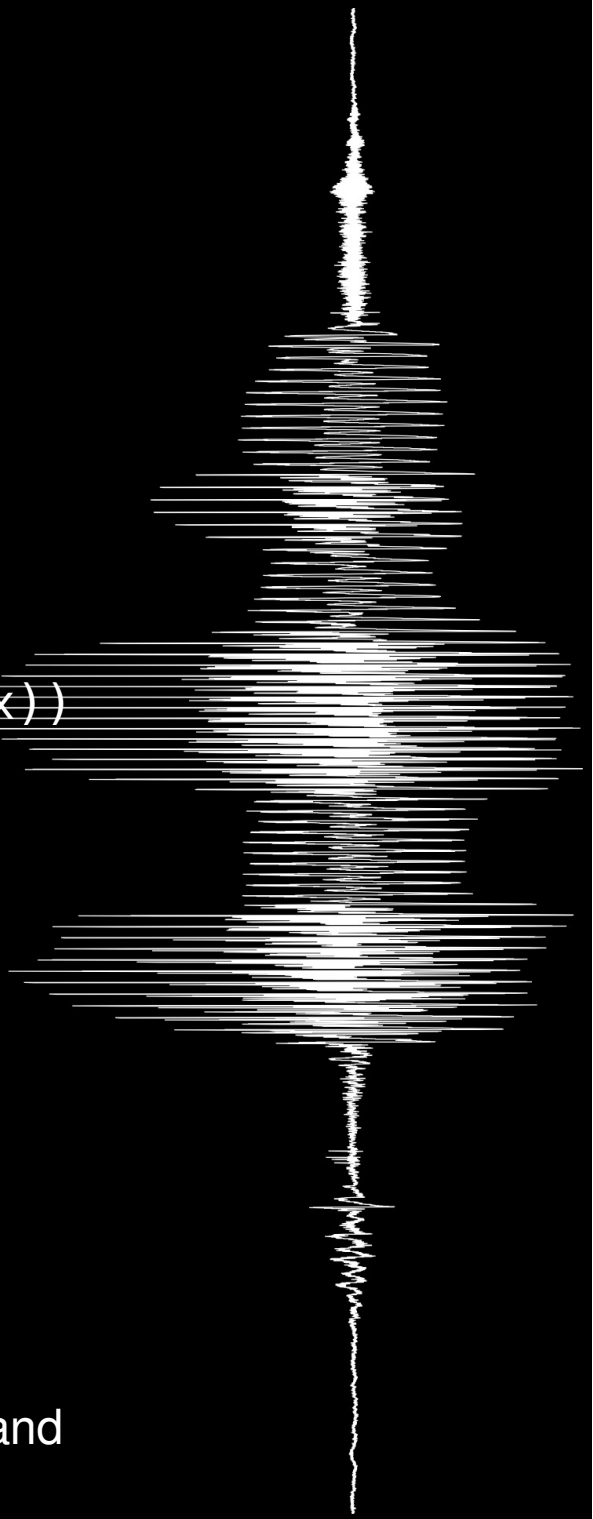
this allows us to

- create objects

```
do("Create Sound from formula...",
   ..."A", 1, 0, 1, 44100, sin(2*pi*440*x))
do("To TextGrid...",
   ..."intervals points”, "points")
```

- remove objects

```
selectObject(1)
do("Remove")*
```



* this is another way to remove objects: using the `"Remove"` command (commands in praat are *case sensitive*!)

the push of a button

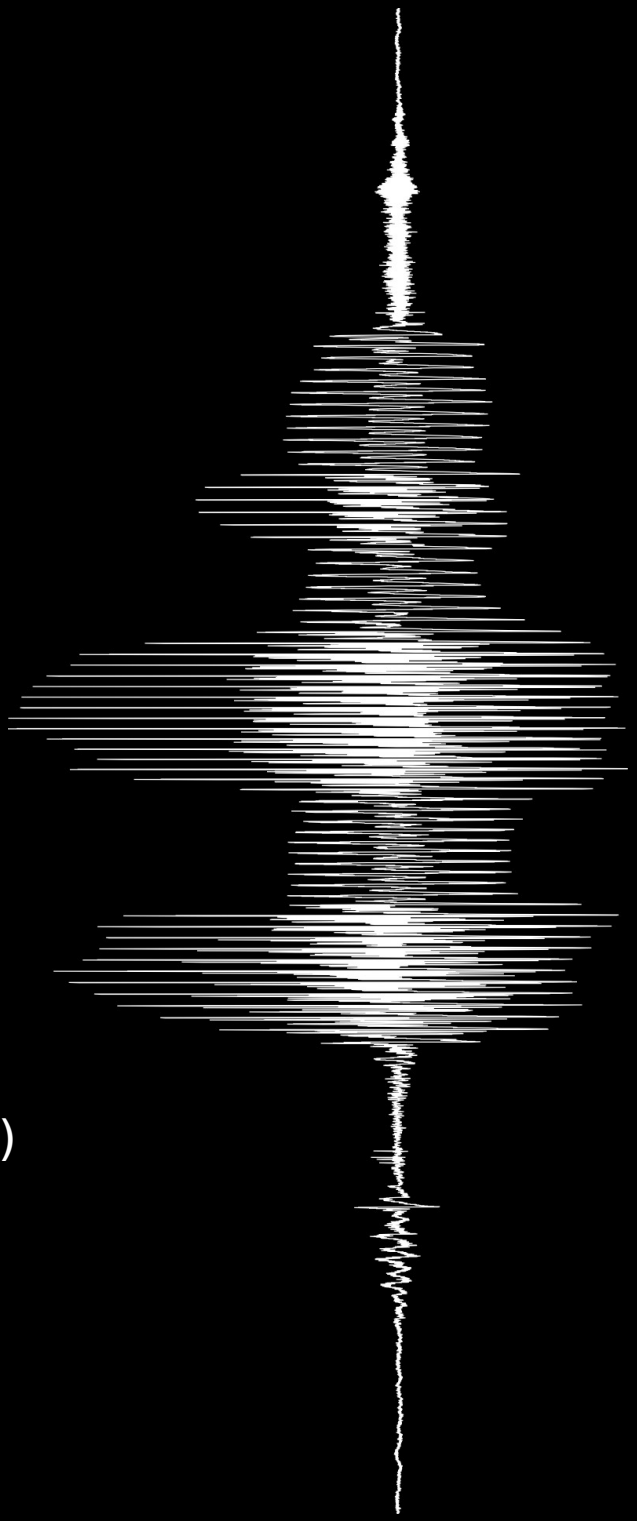
this allows us to

- query information about objects

```
do("Create Sound from formula...",
   ..."A", 1, 0, 1, 44100,
   ...sin(2*pi*440*x))
duration = do("Get total duration")
```

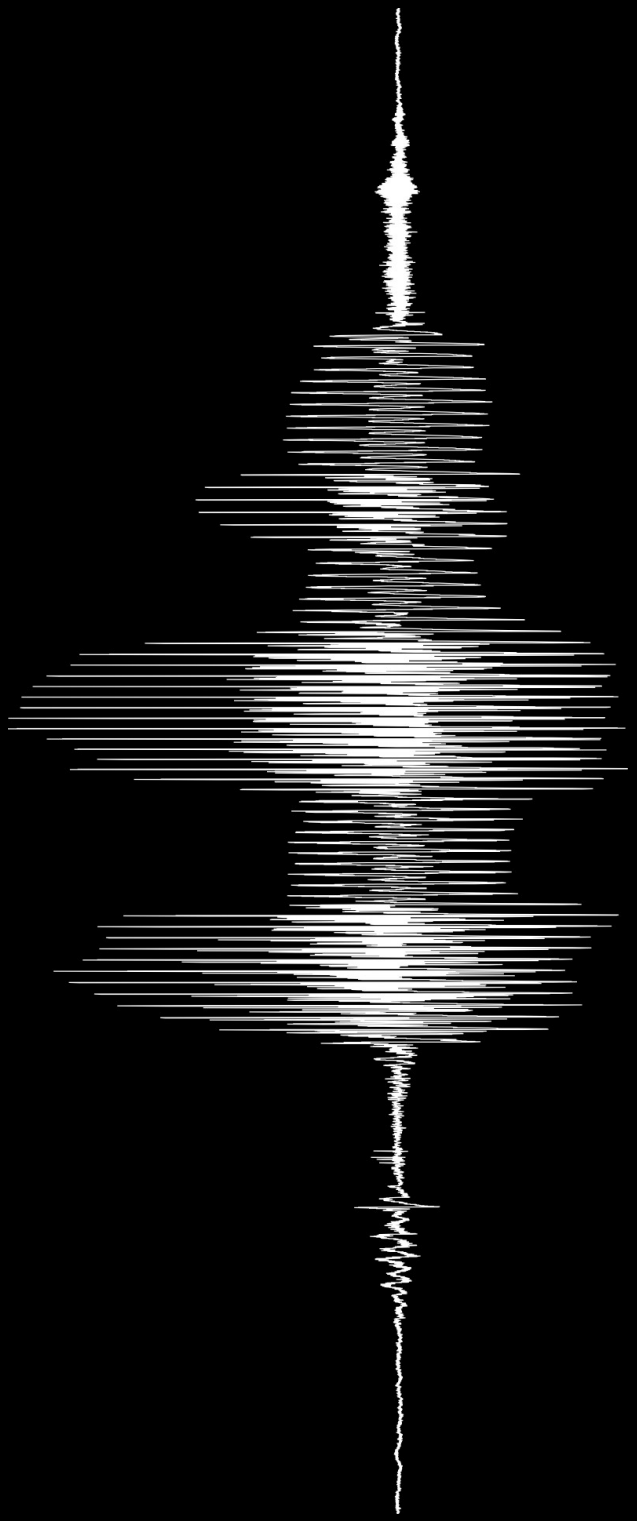
- manipulate objects

```
sound = selected()
quarter = duration / 4
do("To TextGrid...", "points", "points")
for i to 4
   point = quarter * i
   do("Insert point...", 1, "point")
endfor
plus sound
do("Edit")
```



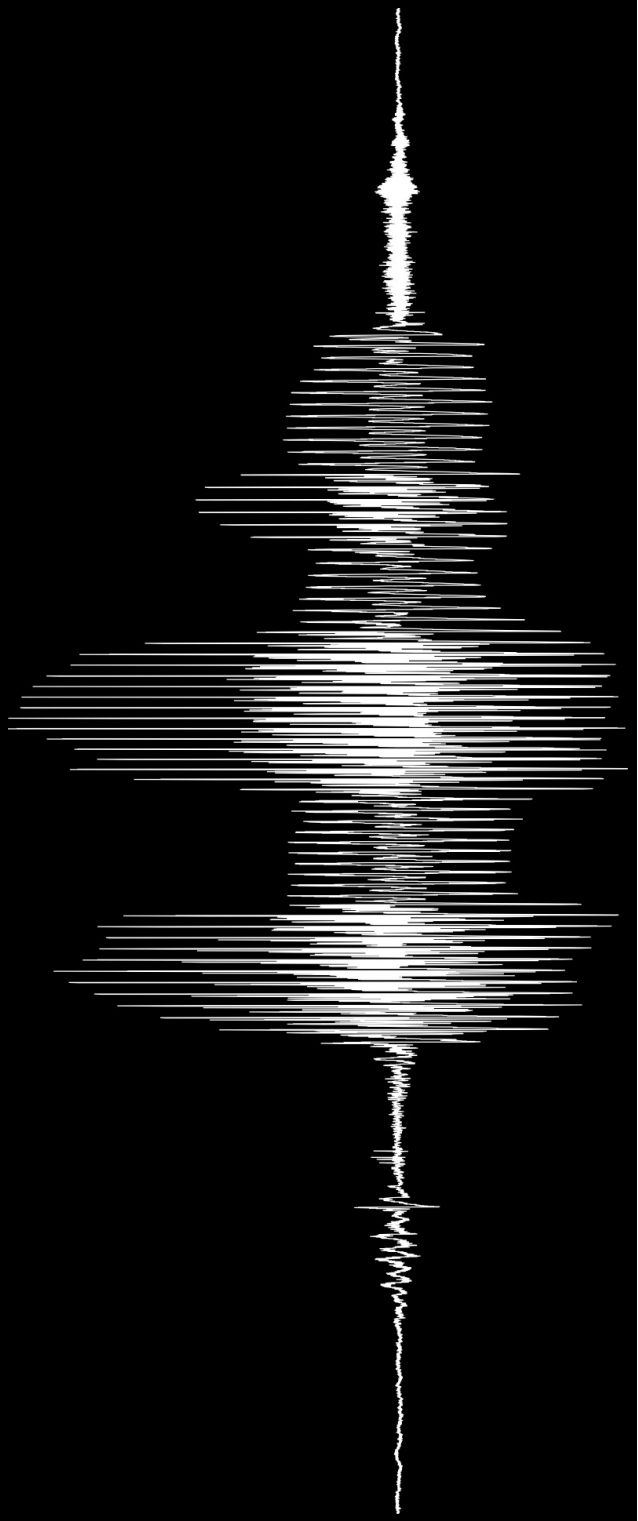
part 5

the tool box



the tool box

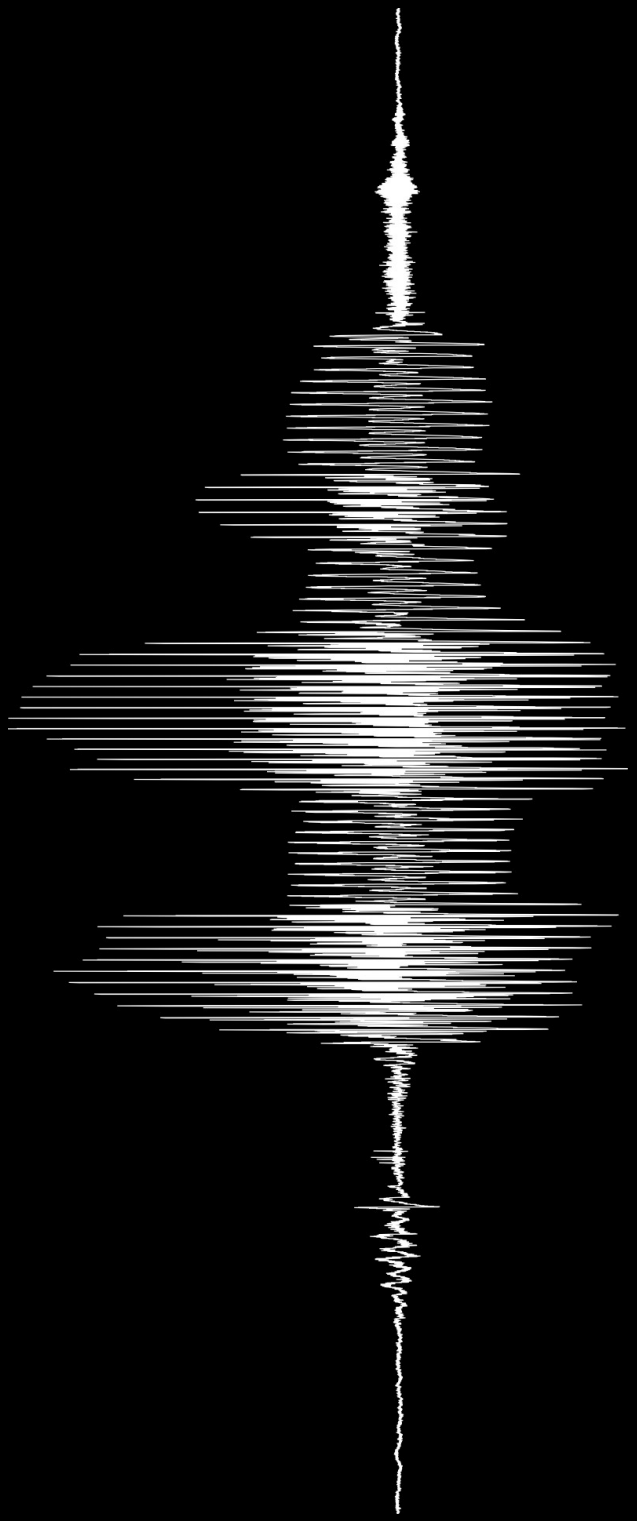
- scripts carry out tasks
- tools make tasks easier
- tools are specialized
- praats tools are its functions



what is a function?

- a function is a ready-made set of instructions that performs a more complex task
- it can be used like a simple instruction
- they can take arguments
- they can return results

    which can, of course, be assigned to a variable



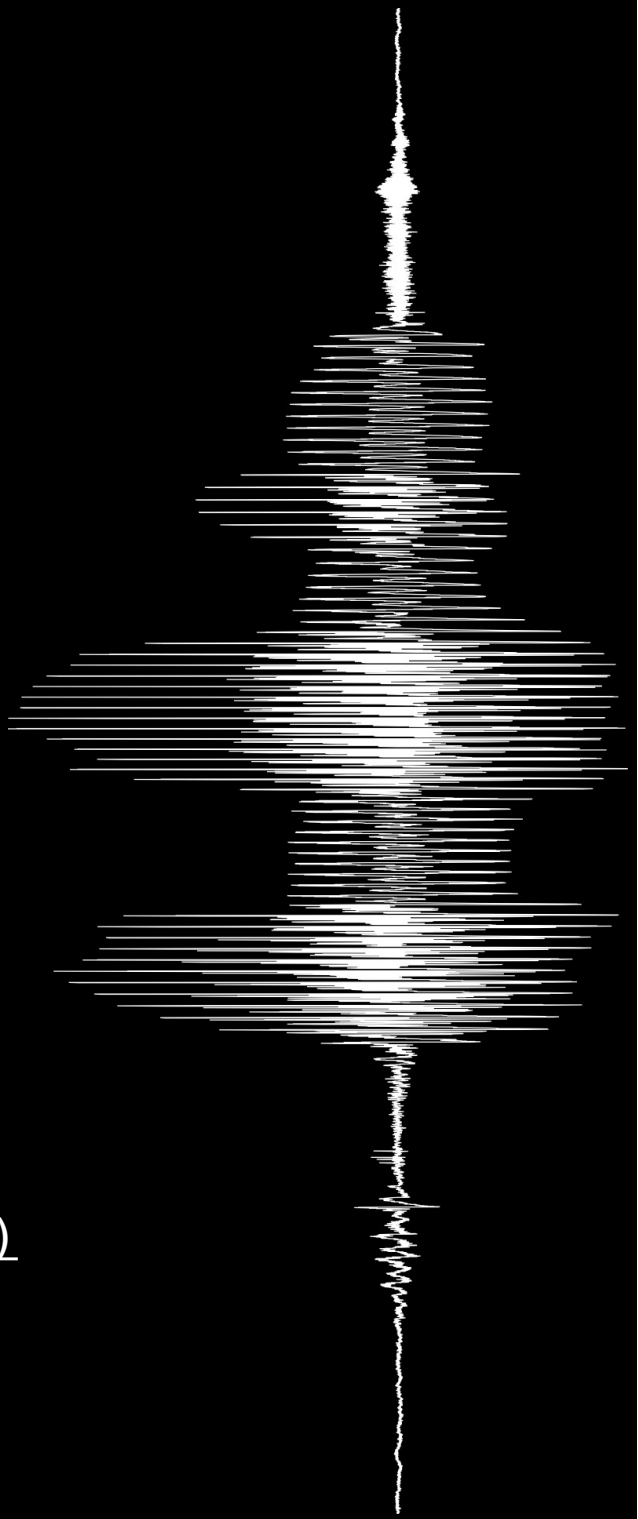
what is a function?

- they are a powerful tool

  a script is a set of simple instructions to perform complex tasks

  a function is just the same!

  we can deal with some complex tasks as if they were simple



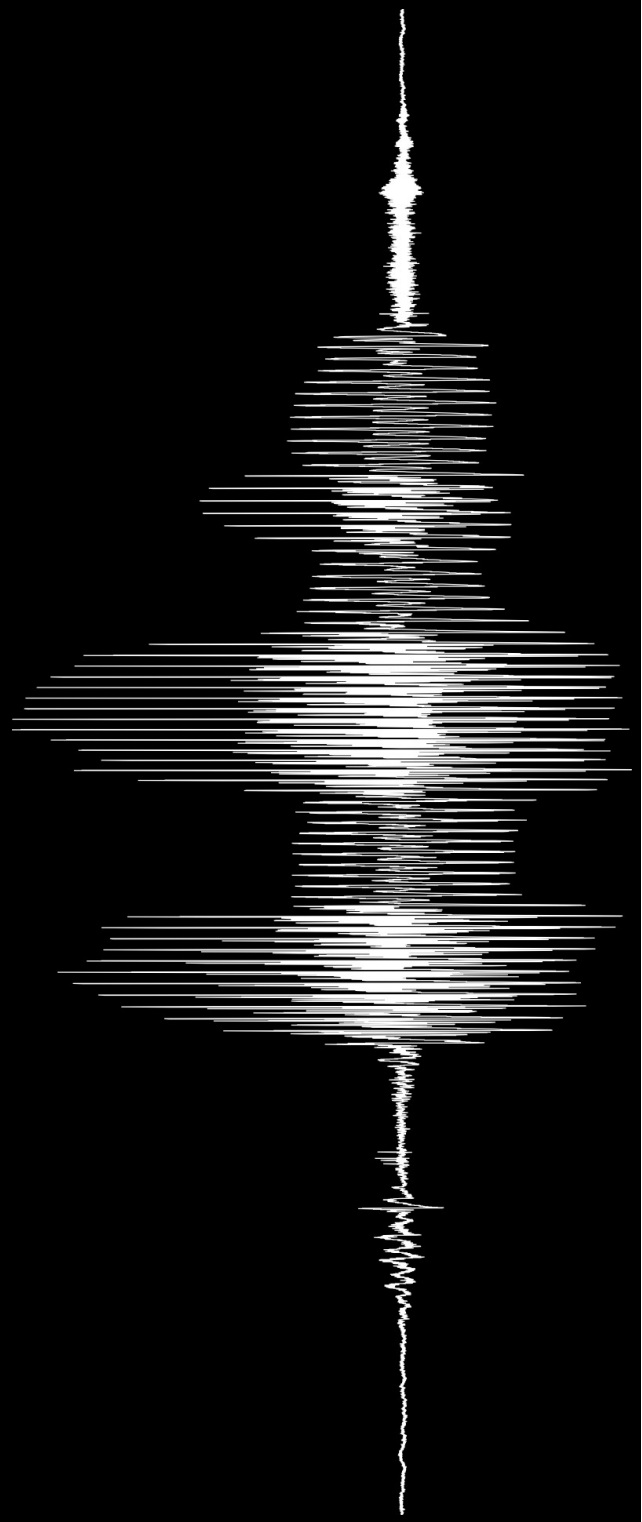
```
do("Create Sound from formula...",
   ..."sound", 1, 0, 1, 44100,
   ...1/2*sin(2*pi*377*x)+randomGauss(0, 0.1))
```

these are functions!

how do you recognize them?

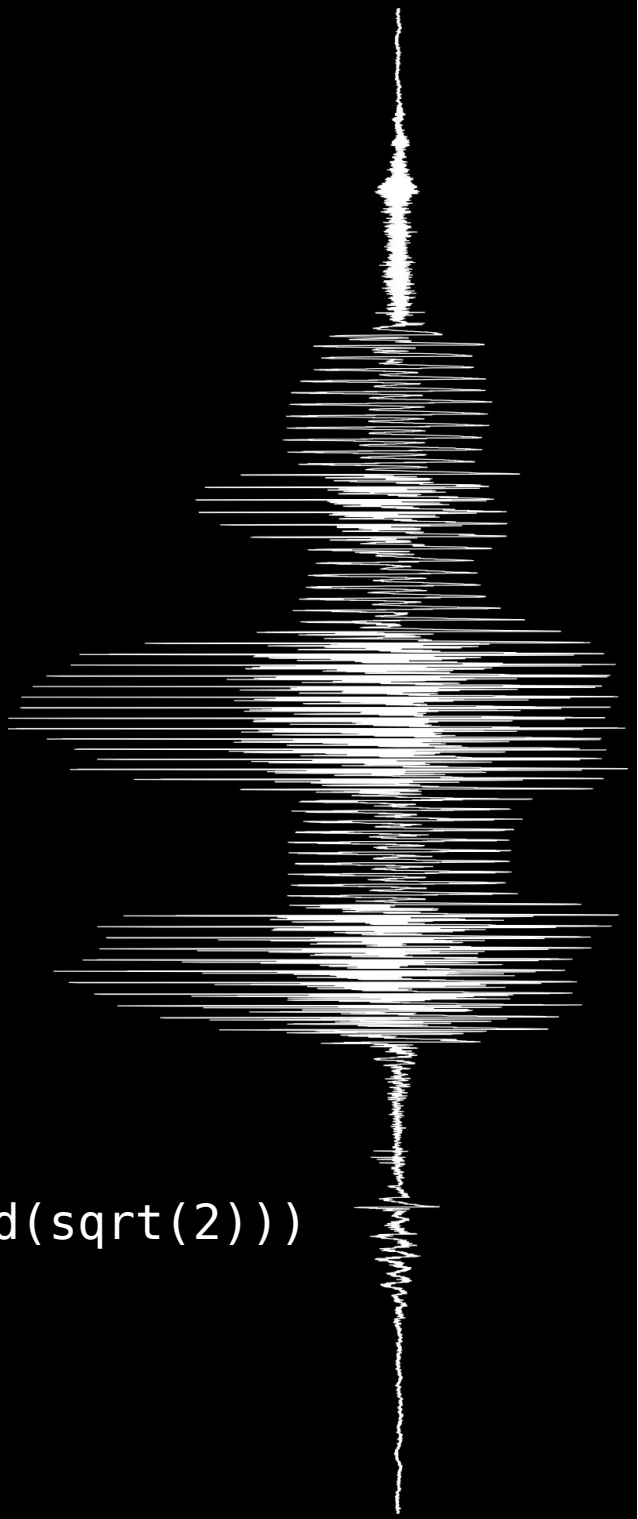
- the easiest way is because they are followed by parenthesis

```
sounds = numberOfSelected("Sound")
```



what's special about them?

- they are invoked by name
- normally at the right side of an assignment (they are *rvalues*)
- they take arguments (which can be the result of other functions)
- they don't always require them



```
root = sqrt(2)
tmp = round(root)
appendInfoLine(tmp)
```

=

```
appendInfoLine(round(sqrt(2)))
```

what's special about them?

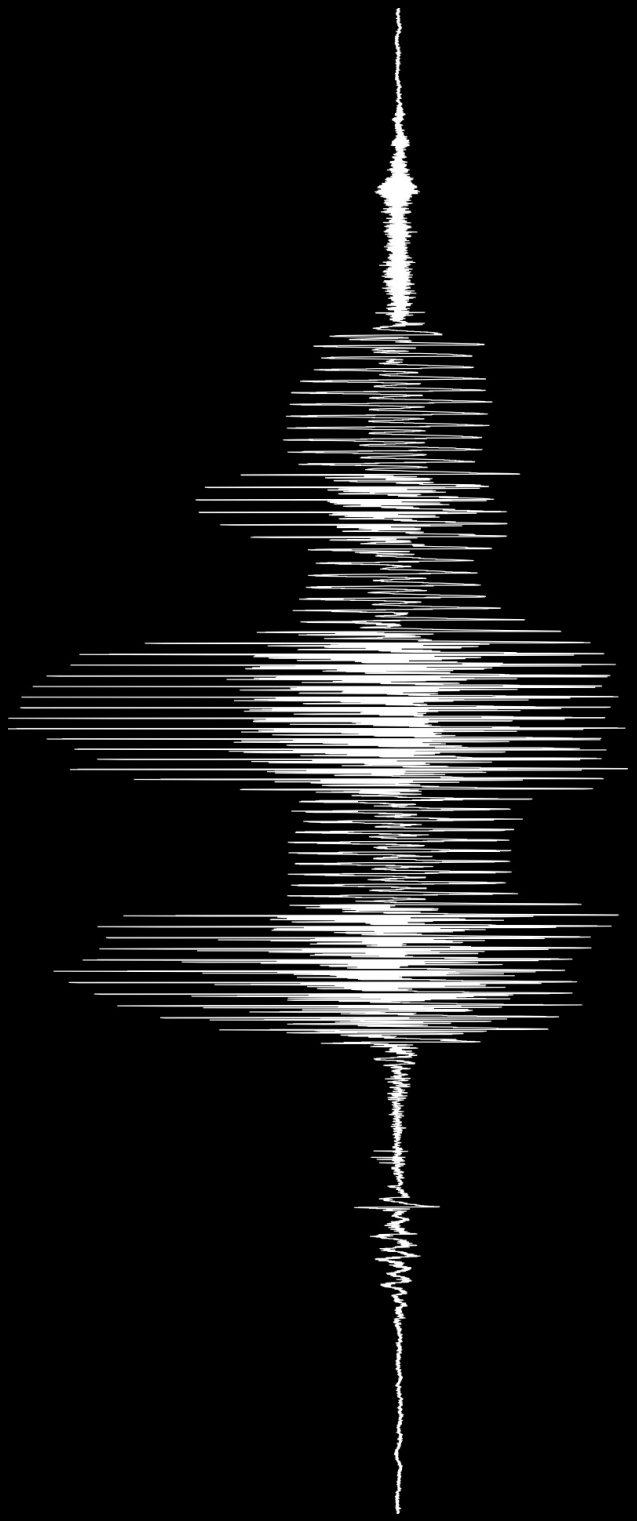
- **all** functions return a value*
- numeric functions return a number, string functions return a string

  pay attention to the name of the function!

```
selected(“Sound”)
```

≠

```
selected$(“Sound”)
```

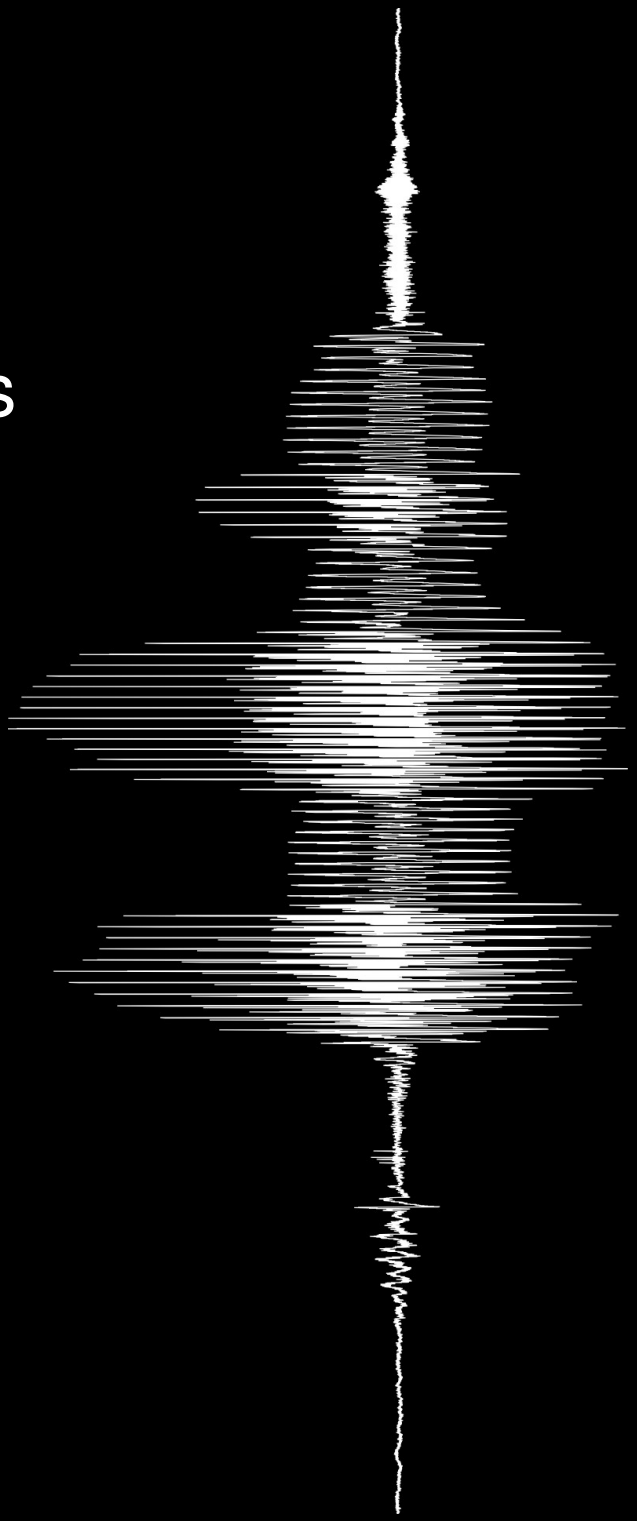


* at least in praat

the usual suspects

we've actually already seen some functions

- `selected(a$, b)`
- `selected$(a$, b)`
- `numberOfSelected(a$)`
- `round(a)`
- `sqrt(a)`

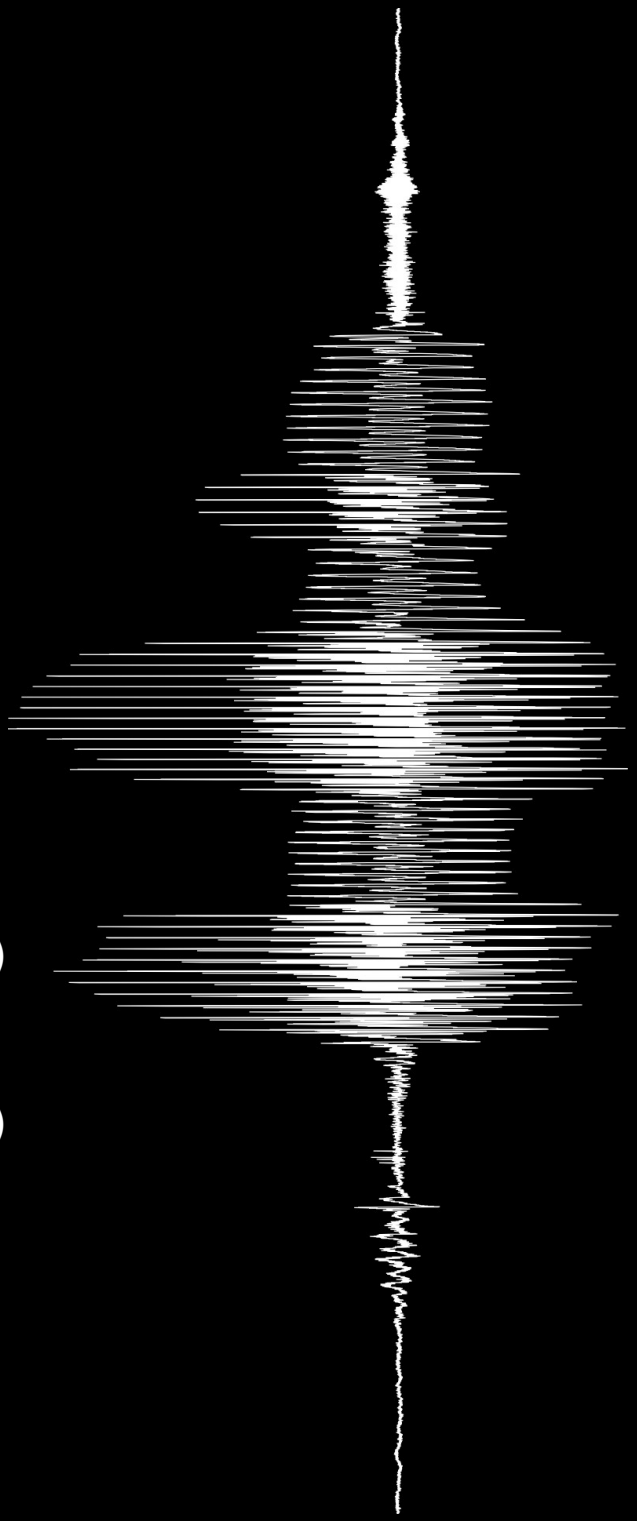


some other useful functions

- `length(a$)`

  returns the length in character of string a$

```
a$ = "any old string"
b$ = "some other longer string"
lengthA = length(a$)
lengthB = length(b$)
if lengthA > lengthB
   appendInfoLine("""", a$,
      ..."""  is longer than """, b$, """")
else
   appendInfoLine("""", b$,
      ..."""  is longer than """, a$, """")
endif
```



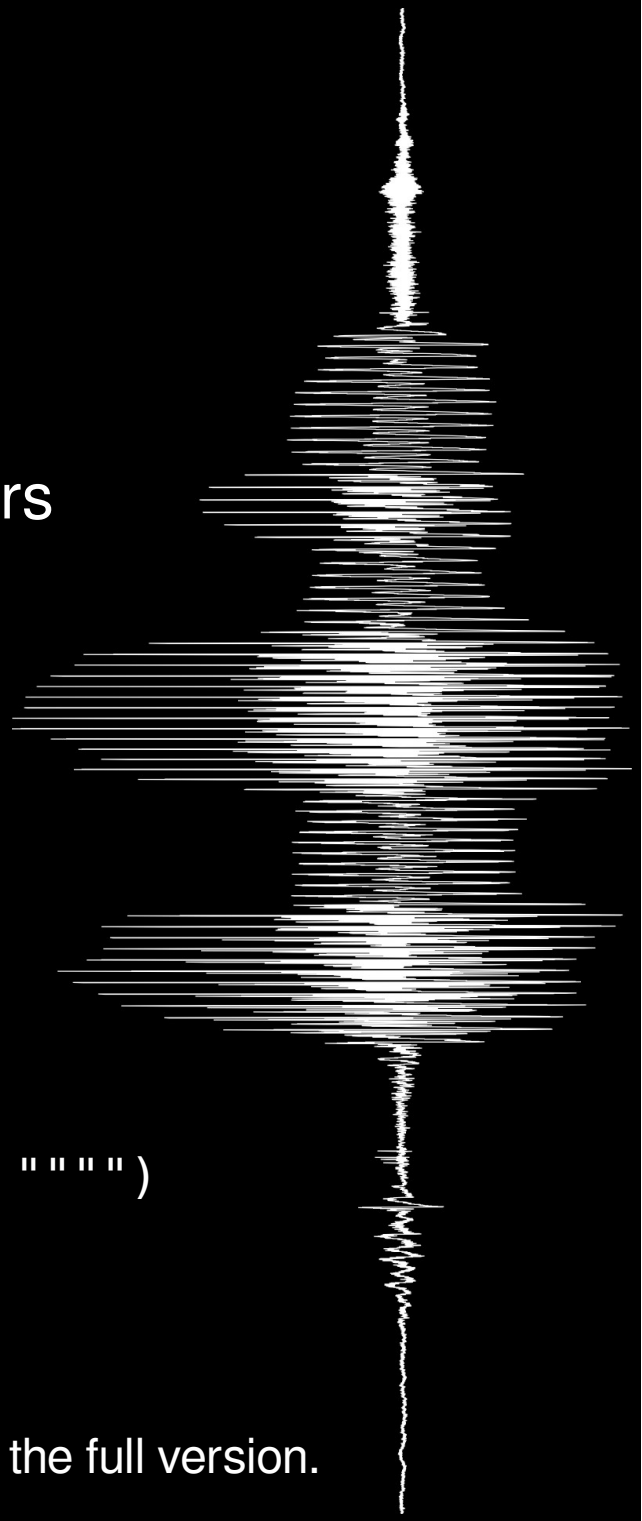
in order to include a double quotation character in a string,
you need to input it twice

some other useful functions

- `left$(a$, b); right$(a$, b)`

  return a string of length b made of characters on the left or right side of a$

```
[...]*
for i to numberOfTokens
    initial$ = left$(token$[i], 1)
    ending$ = right$(token$[i], 1)
    appendInfoLine(token$[i],
        ..." begins with an """, initial$,
        ..."""" and ends with an """, ending$, """")
endfor
```

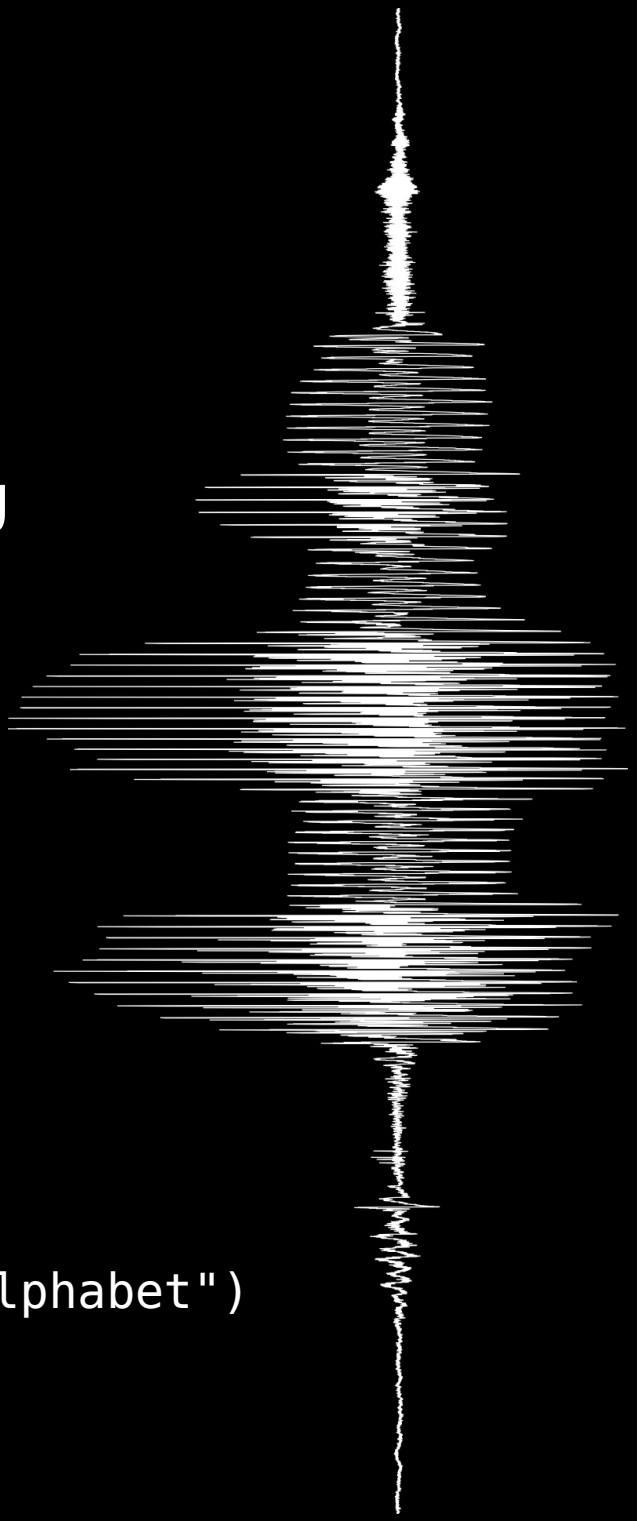


* this denotes an edited part of the script. see the example scripts for the full version.

some other useful functions

- `mid$(a$, b, c)`

  returns a substring of `a$` of length `c` starting from position b in `a$`

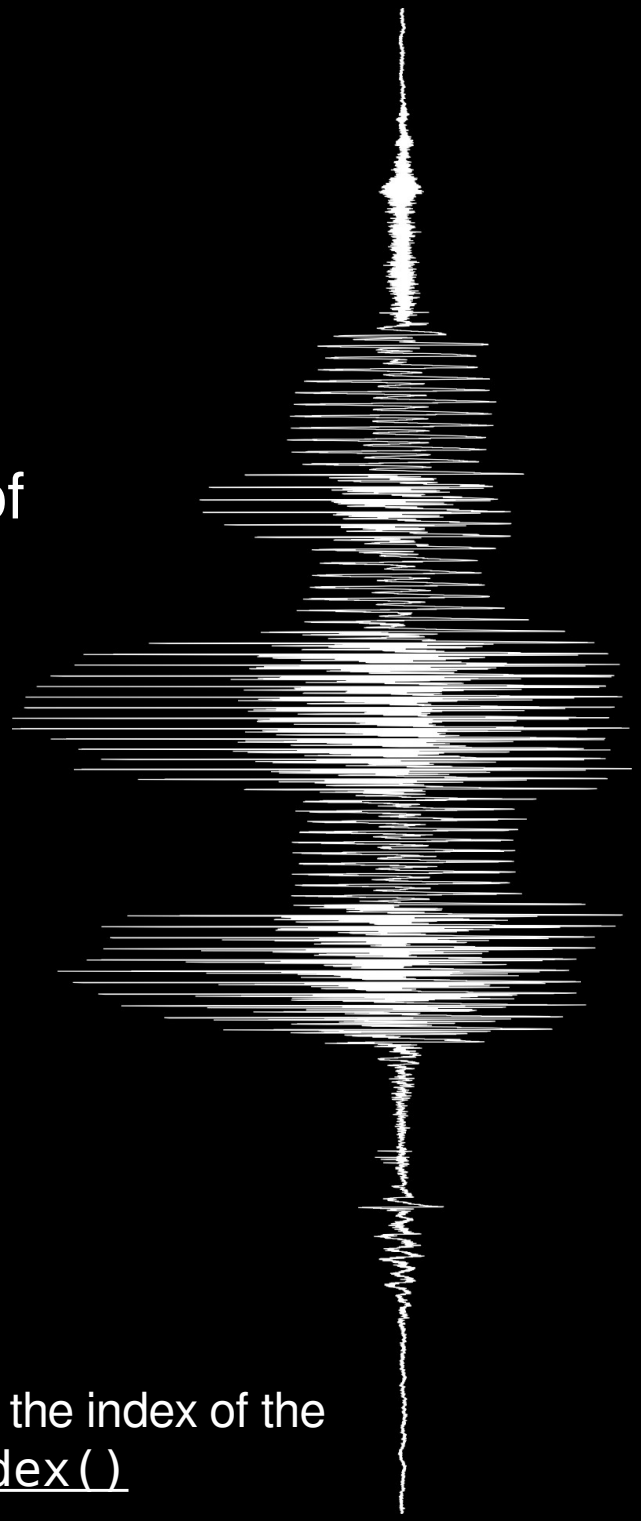


```
vowels$ = "aeiou"
strlen = length(vowels$)
a$ = ""
for i to strlen
    a$ = a$ + mid$(vowels$, i, 1)
    if i < strlen
        a$ = a$ + "-"
    endif
endfor
appendInfoLine(a$, " are the vowels in the alphabet")
```

some other useful functions

- `index(a$, b$)`

  returns the position of the first occurrence of `b$` in `a$`; returns `0` if `b$` cannot be found in `a$`



```
n = do("Get number of intervals...", 1)
for i to n
    vowels$ = "aeiou"
    label$ = do$("Get label of interval...",
        ...1, i)
    if index(vowels$, label$) > 0
        appendInfoLine("Is vowel: ", label$)
    endif
endfor
```

`index()` has a sister function that returns the index of the *last* occurrence of a string in another: `rindex()`

some other useful functions

- `fixed$(a, b)`

  return a string with the value of `a`, with `b` digits after the decimal point

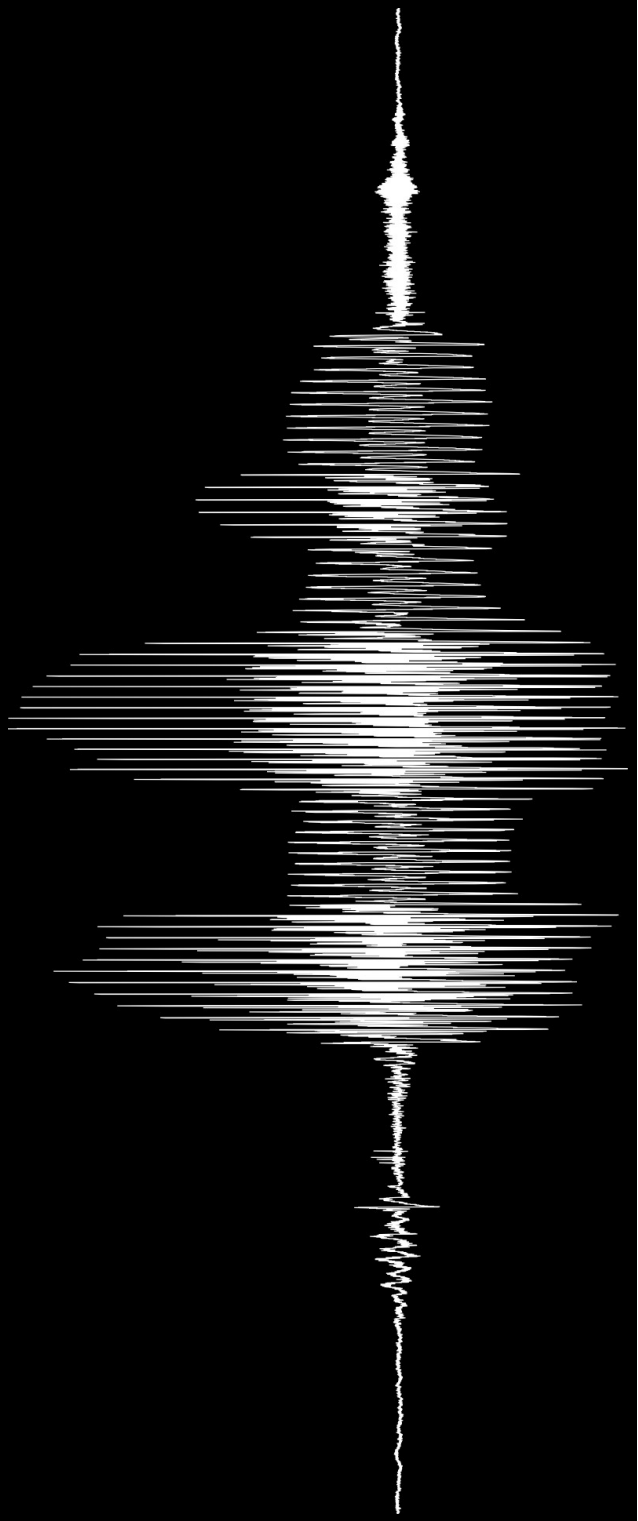
- `number(a$)`

  interprets `a$` as a number

  (these can be used to turn numbers into strings and vice-versa)

- `date()`

  returns the current date in a preset format



some other useful functions

- `extractNumber(a$, b$)`

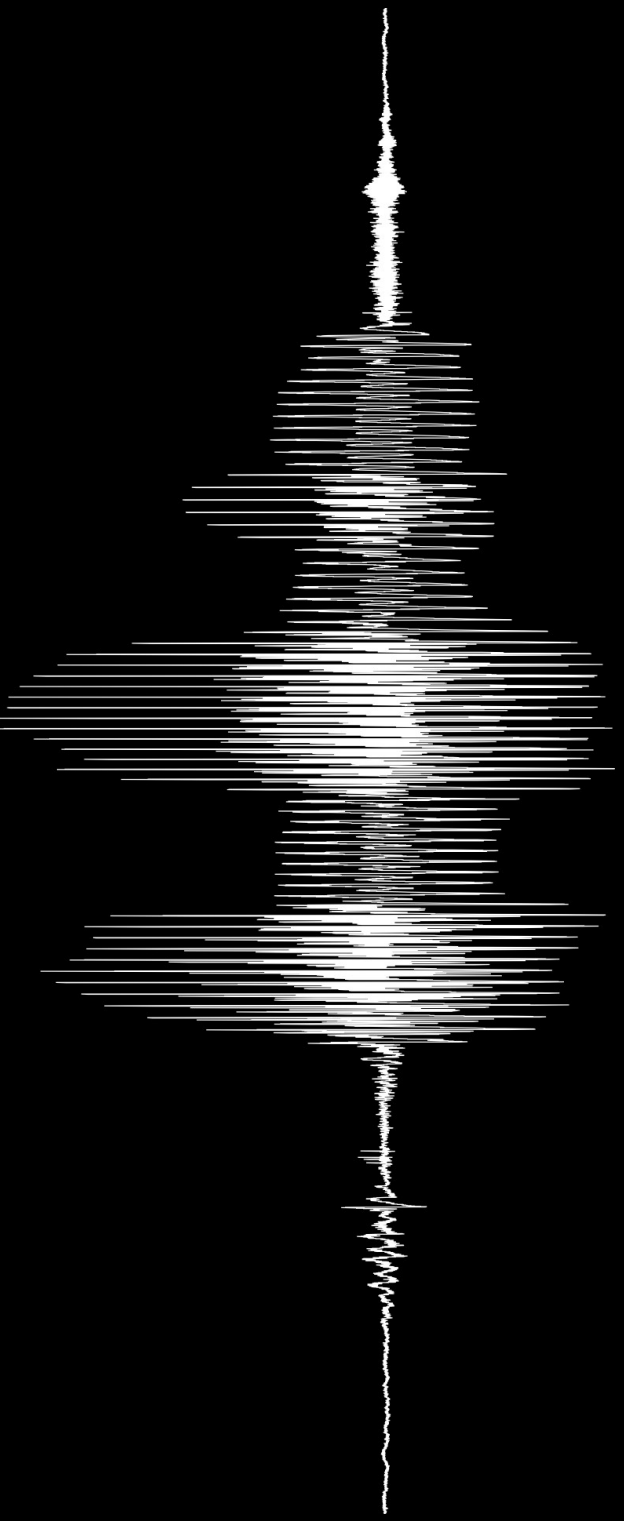
  returns the number in `a$` that follows `b$`

- `extractWord$(a$, b$)`

  returns a string with the word (no spaces) in `a$` that follows `b$`

- `extractLine$(a$, b$)`

  returns a string with whatever is in `a$` between `b$` and the next line break



some other useful functions

- `hertzToSemitones(a)`

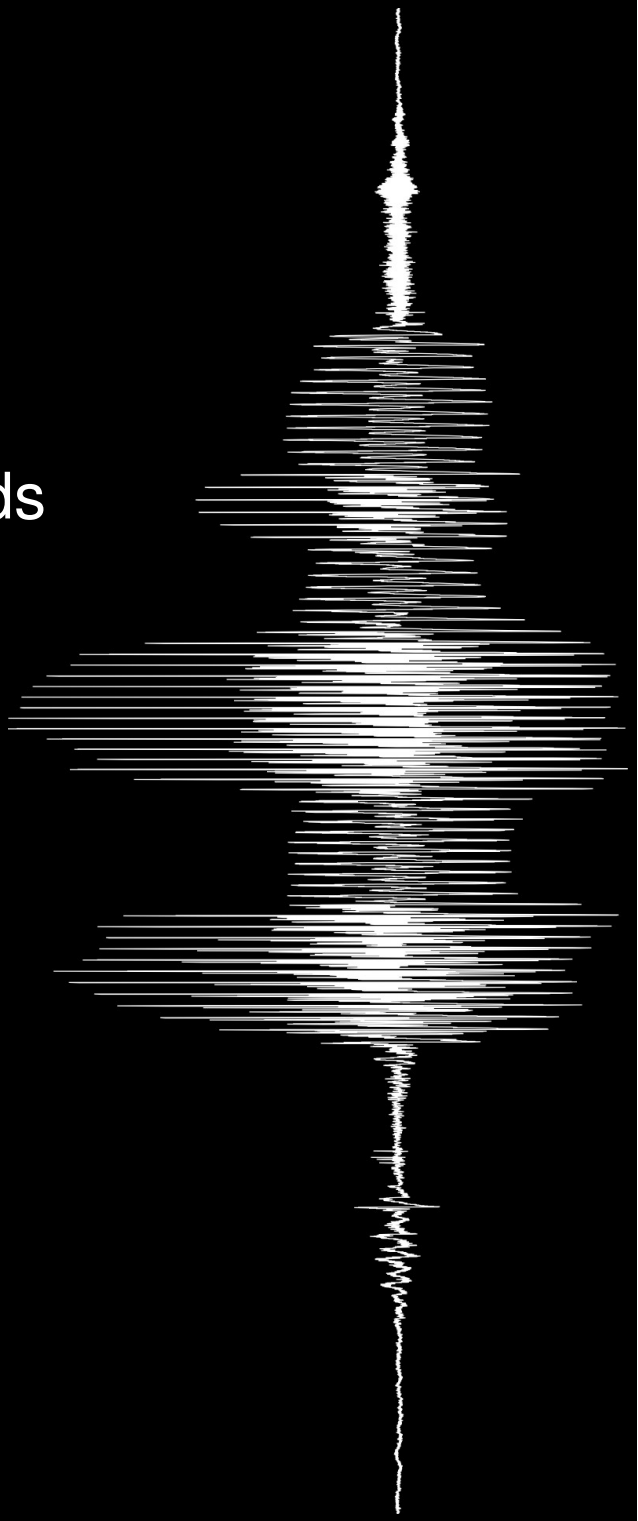
  returns a value in semitones that coresponds to a in Hz (relative to 100Hz)

- `semitonesToHertz(a)`

  returns the result of the inverse operation

- `abs(a)`

  returns the absolute value of a (unsigned magnitude)



and if the function doesn't exist?

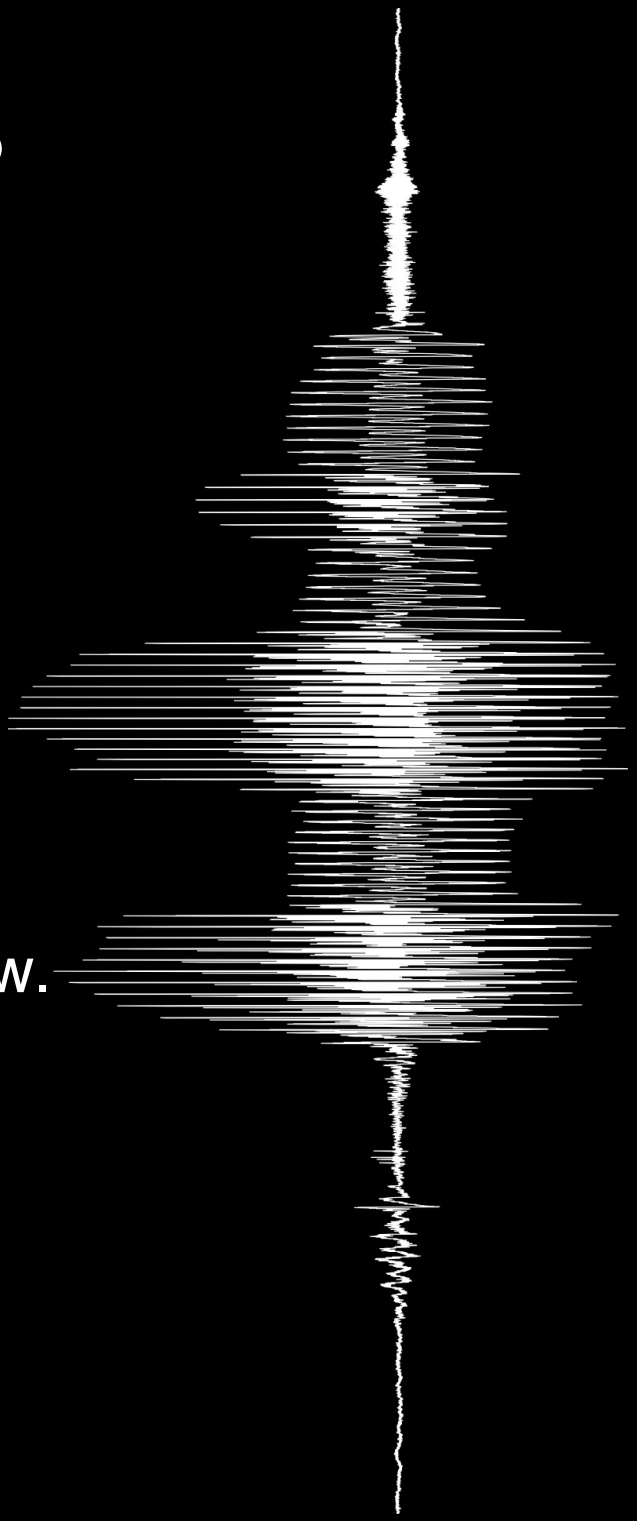
you can make your own!

you can either:

- write it as a procedure inside your script
- write it as a script

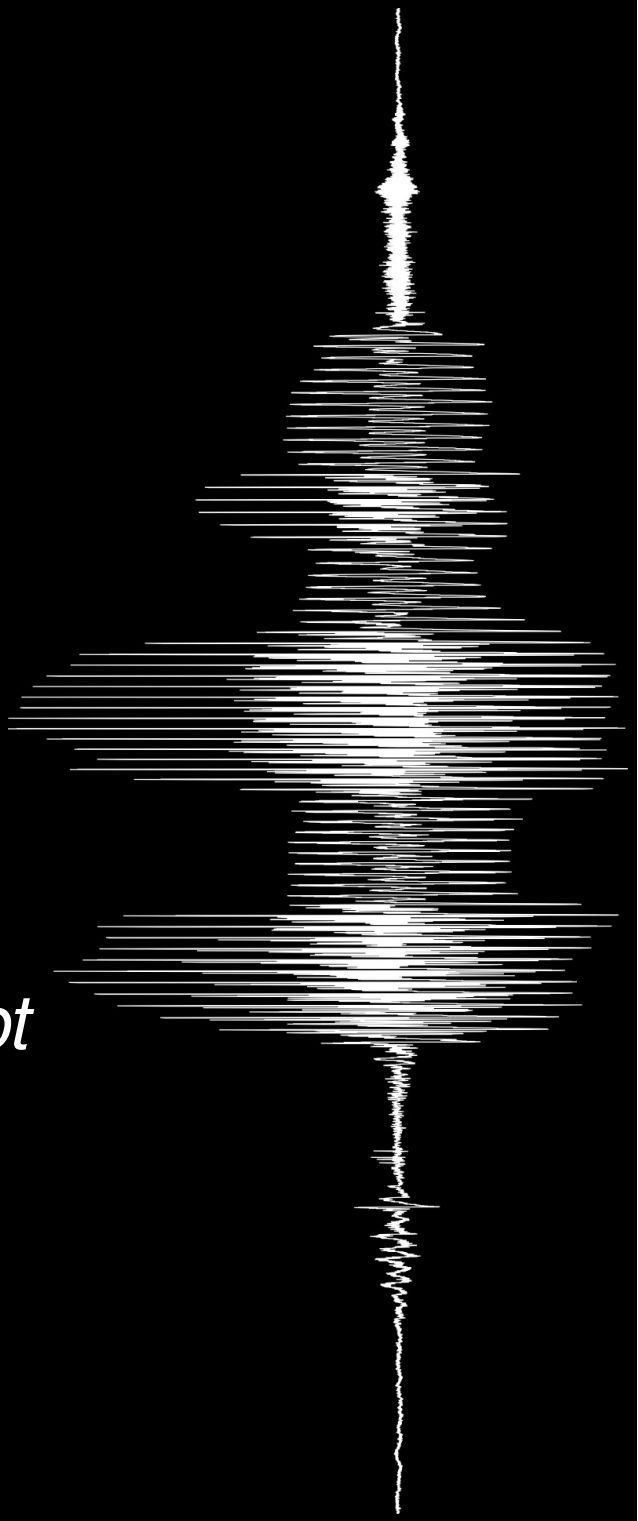
    and then use it with `include` or `execute`...
    but this is more advanced than this overview.
    feel free to check it on the praat manual

    (follow those links!)



what is a procedure?

- to perform a repetitive task, you use a script
- to perform a repetitive task *in a script*, you can use a procedure
- a procedure is like a *script within a script*
- but they are not functions!



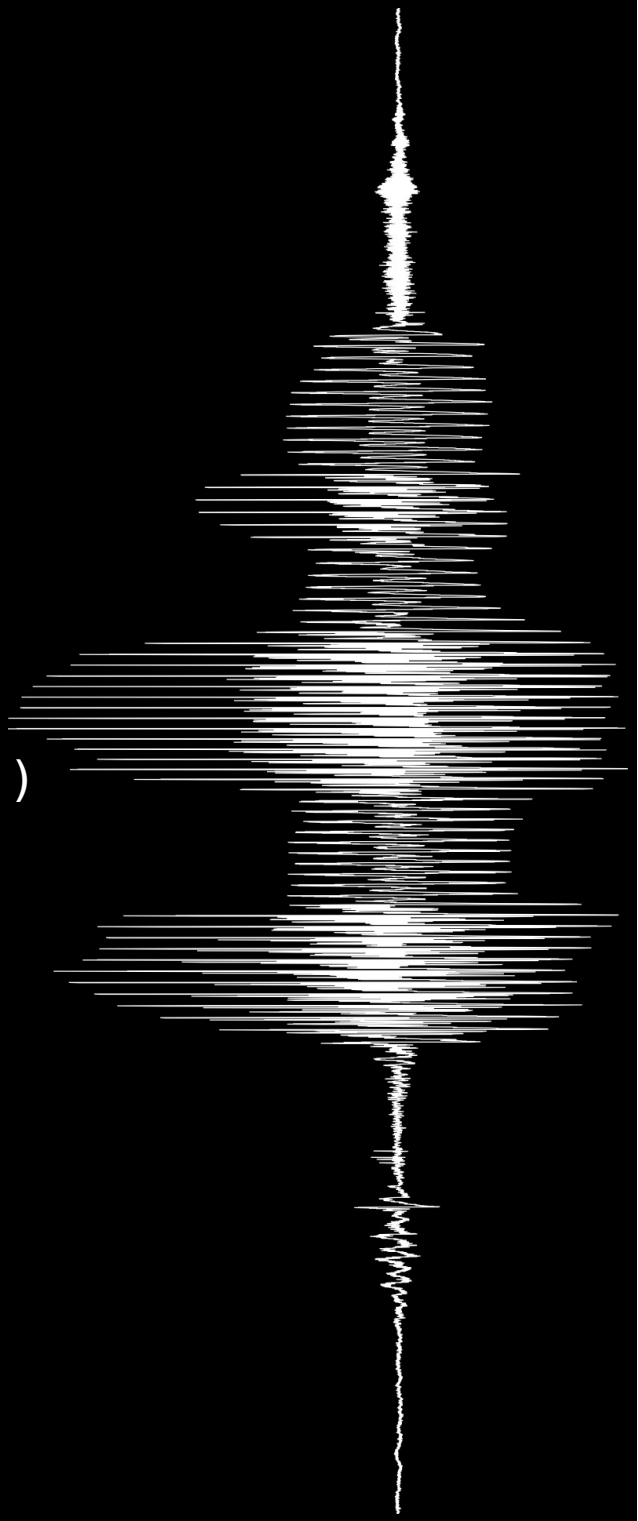
how do you write one?

- you declare it

```
procedure NameOfTheProcedure ([variables])
…
endproc
```

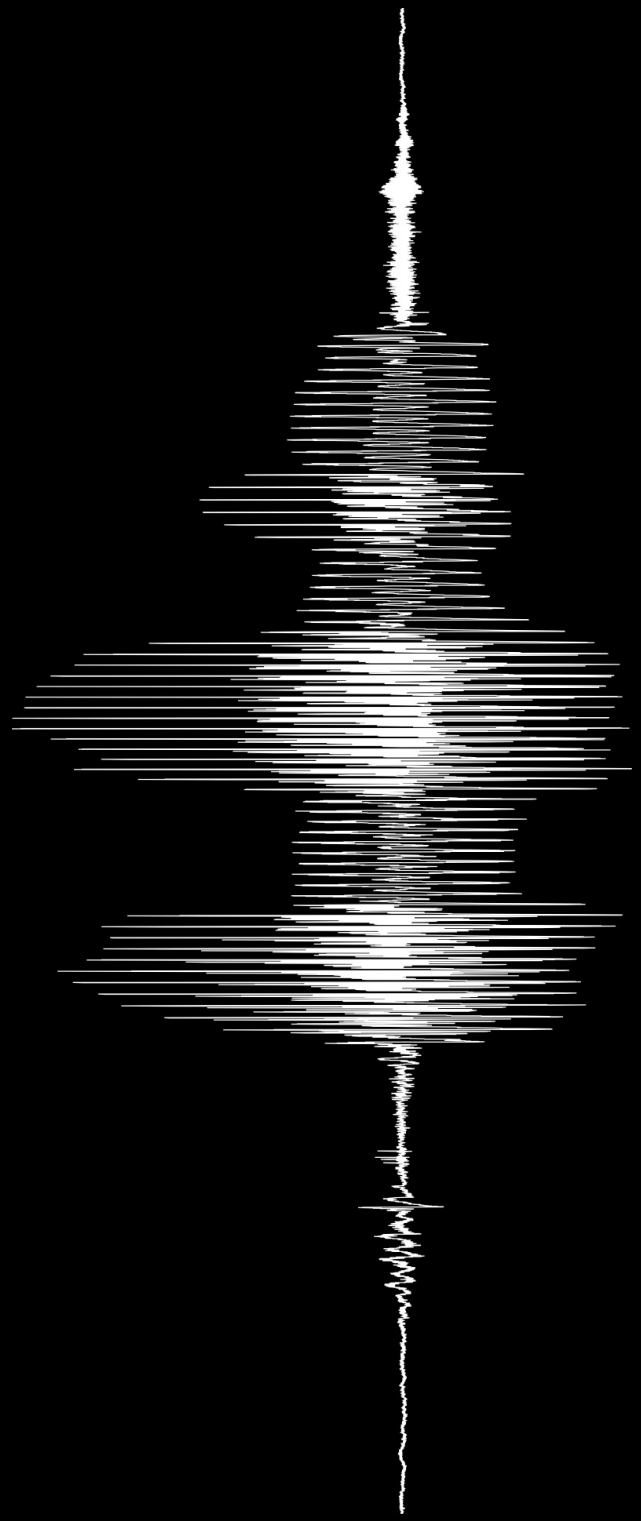
- you invoke it

```
@NameOfTheProcedure([variables])
```



example 1

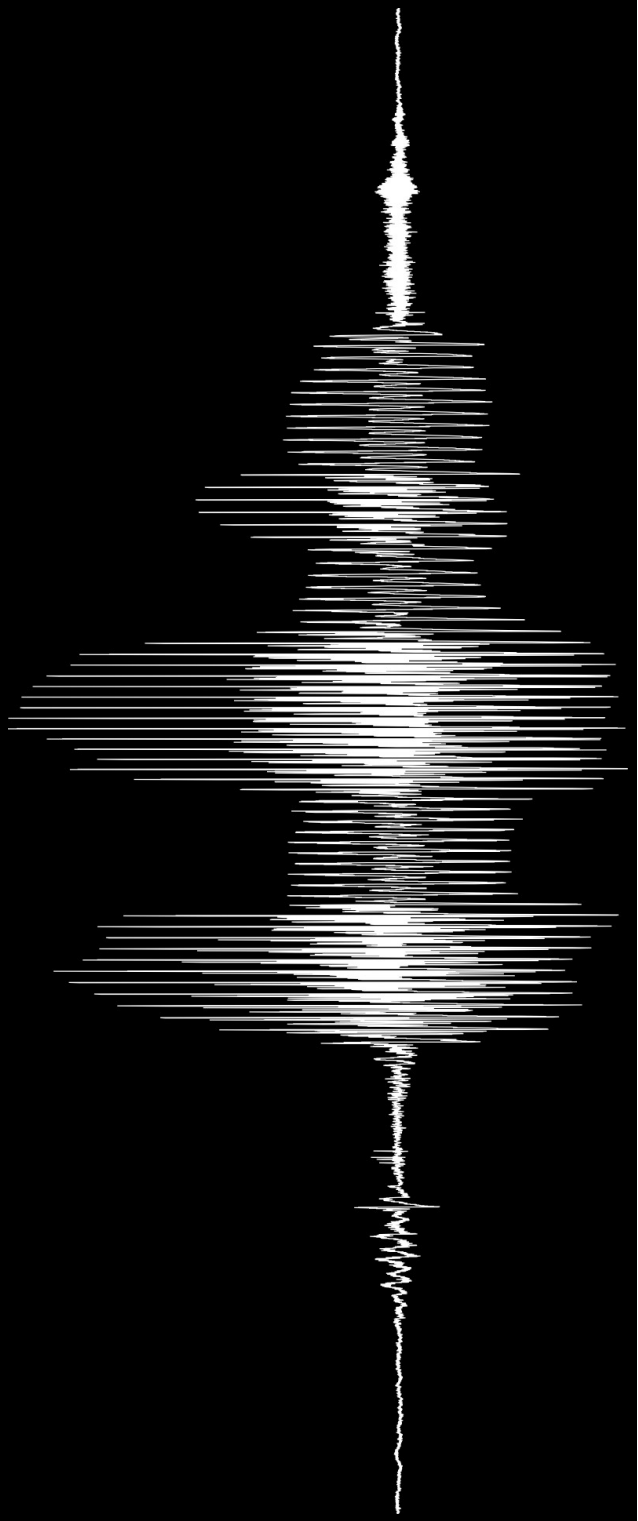
```
word$ = "one"
@measure()
word$ = "two"
@measure()
word$ = "three"
@measure()

procedure measure ()
    strlen = length (word$)
    appendInfoLine(word$ ," is ", strlen,
        ..." characters long")
endproc
```



but functions take arguments...

example 1

- `word$` is a *global variable*, available throughout the script
- `@measure()` was written in its most basic form: it is the same as if the contents of the procedure were copied every time we call it



example 2

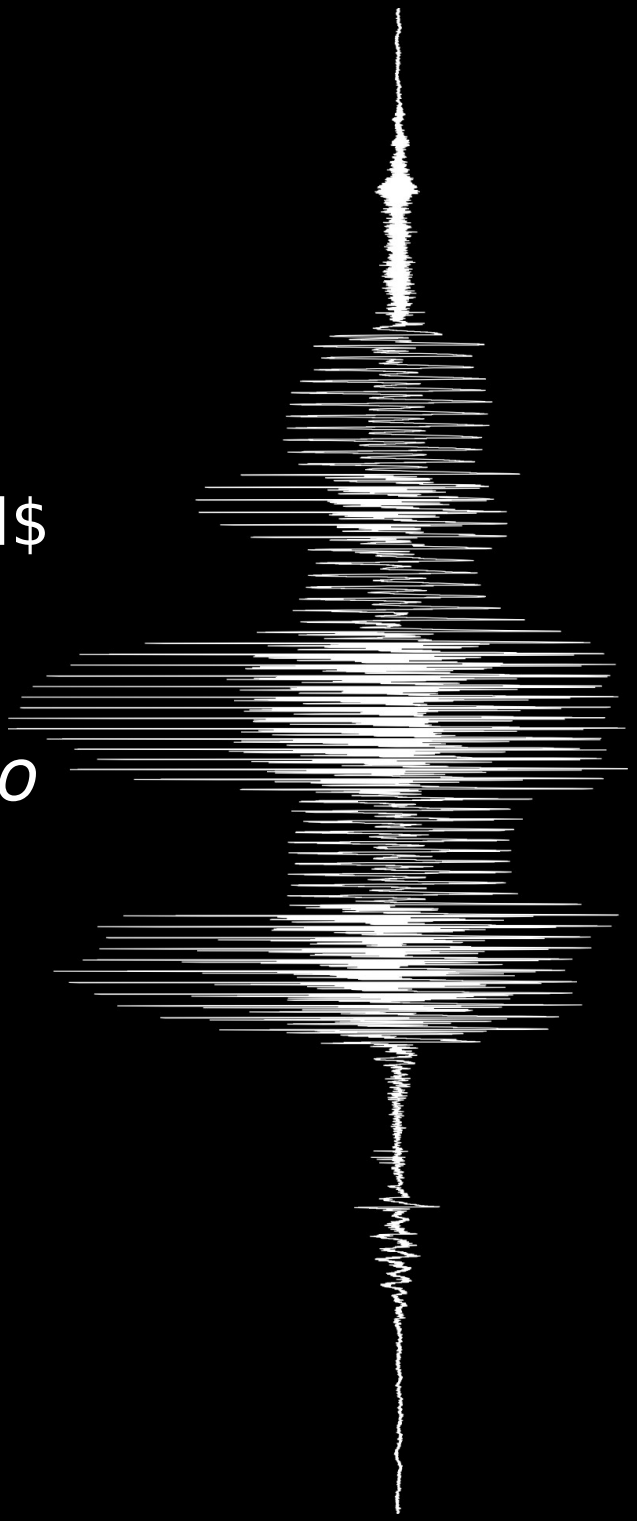
```
word$ = "my word stays the same"
appendInfoLine("word$ = """, word$, """")
@measure("one")
@measure("two")
@measure("three")
appendInfoLine("word$ = """, word$, """")

procedure measure (.word$)
   strlen = length(.word$)
   appendInfoLine("""", .word$, """ is ", strlen,
       ..." characters long")
endproc
```

and what about *return values*?

example 2

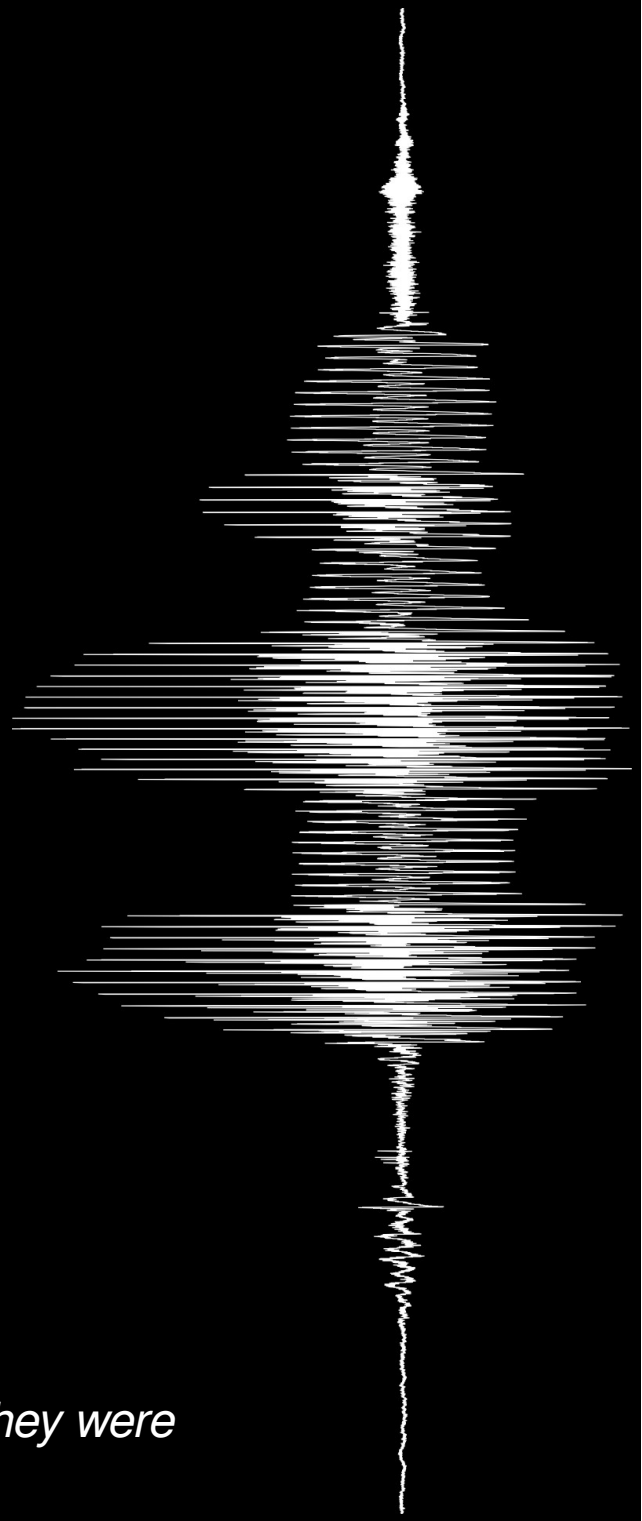
- `word$` is still global
- but now `@measure()` works with `.word$`
- `.word$` is a *local variable*
- however, in praat local variables are *also* available globally

  (`.word$` is the same as `measure.word$`)
- local variables are used to control the information available to procedures
- so how do we get values back?



example 3

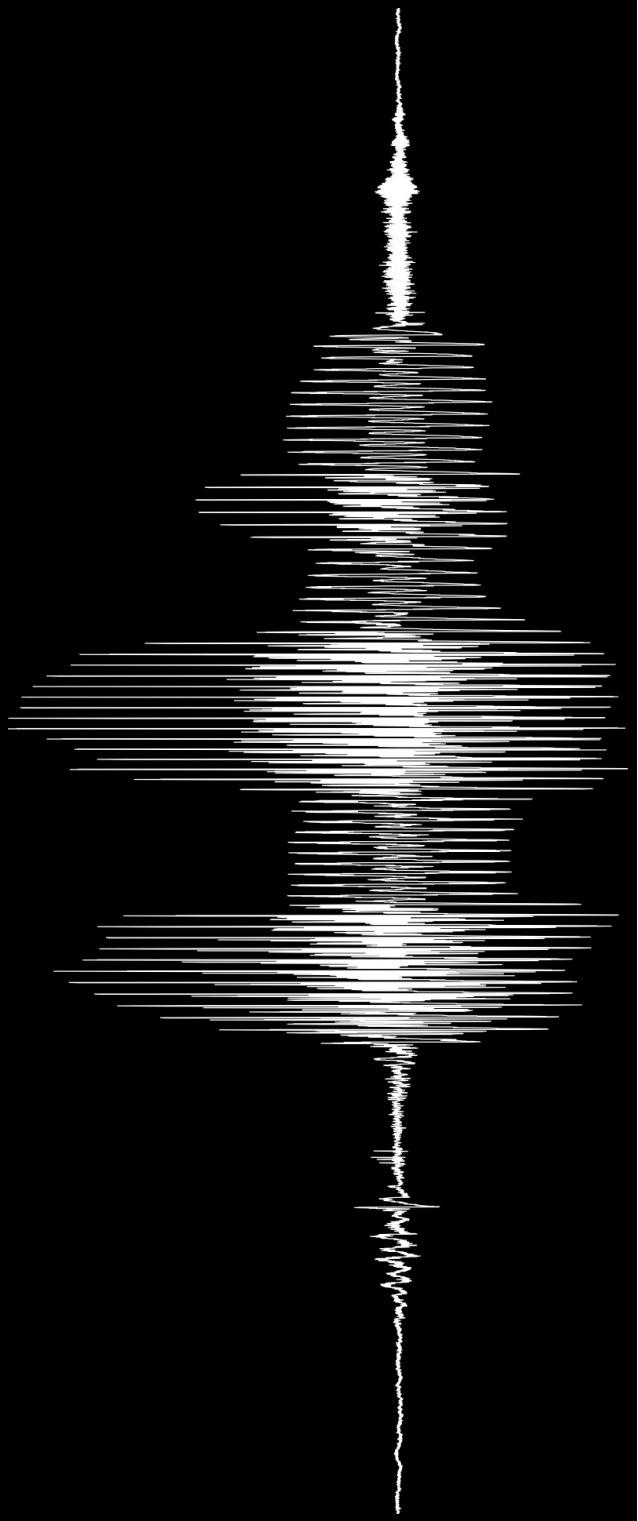
```
vowels$ = "aeiou"
for i to length(vowels$)
   @getChar(i, vowels$)
   appendInfoLine("Vowel ", i, " is """,
      ...getChar.result$, """")
endfor

procedure getChar (.index, .string$)
   .result$ = mid$(.string$, .index, 1)
endproc
```



they may *not be* functions (and they are not),
but that doesn't mean they can't be used *as if they were*

part 6

reaching out

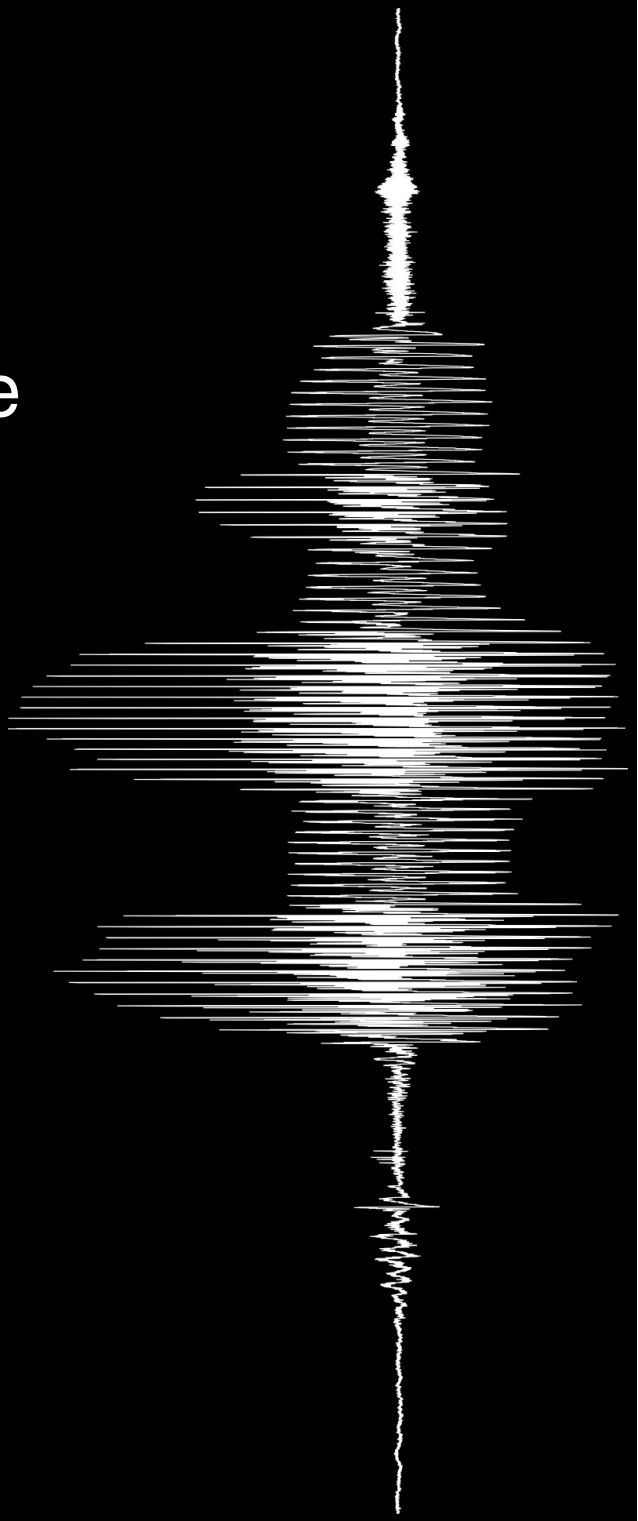


interacting with the user

we also have tools to communicate with the user

mainly, they are:

- `appendInfo()` and `appendInfoLine()`
- `exit`
- `pause` and `form`



through the Info screen

- `appendInfo(string$)`

  `appendInfoLine(string$)`

  `writeInfoLine(string$)`

  `appendInfo()` and `appendInfoLine()` add to the Info screen without clearing it, but the latter always ends with a line break

  `writeInfoLine()` clears the Info screen and outputs a line of text to it

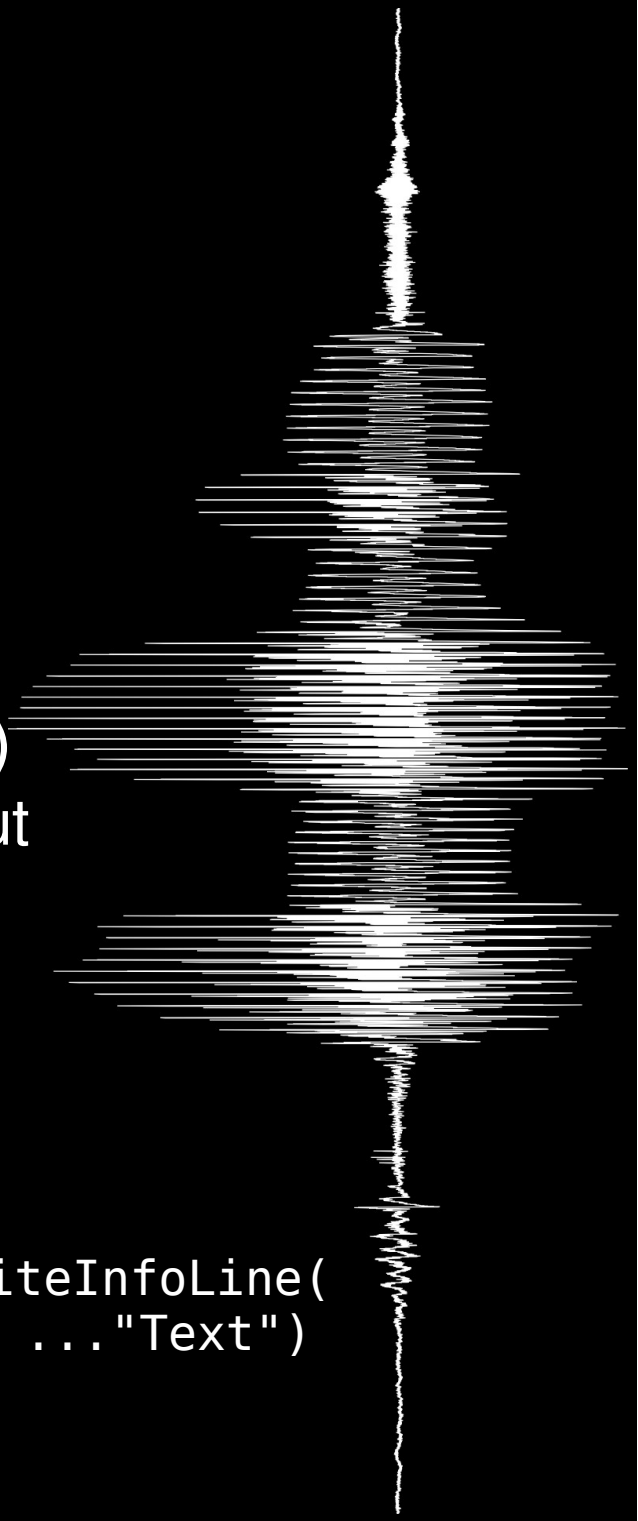
```
clearinfo
appendInfo(
    ..."Text",
    ...newline$)
```

=

```
clearinfo
appendInfoLine(
    ..."Text")
```

=

```
writeInfoLine(
    ..."Text")
```



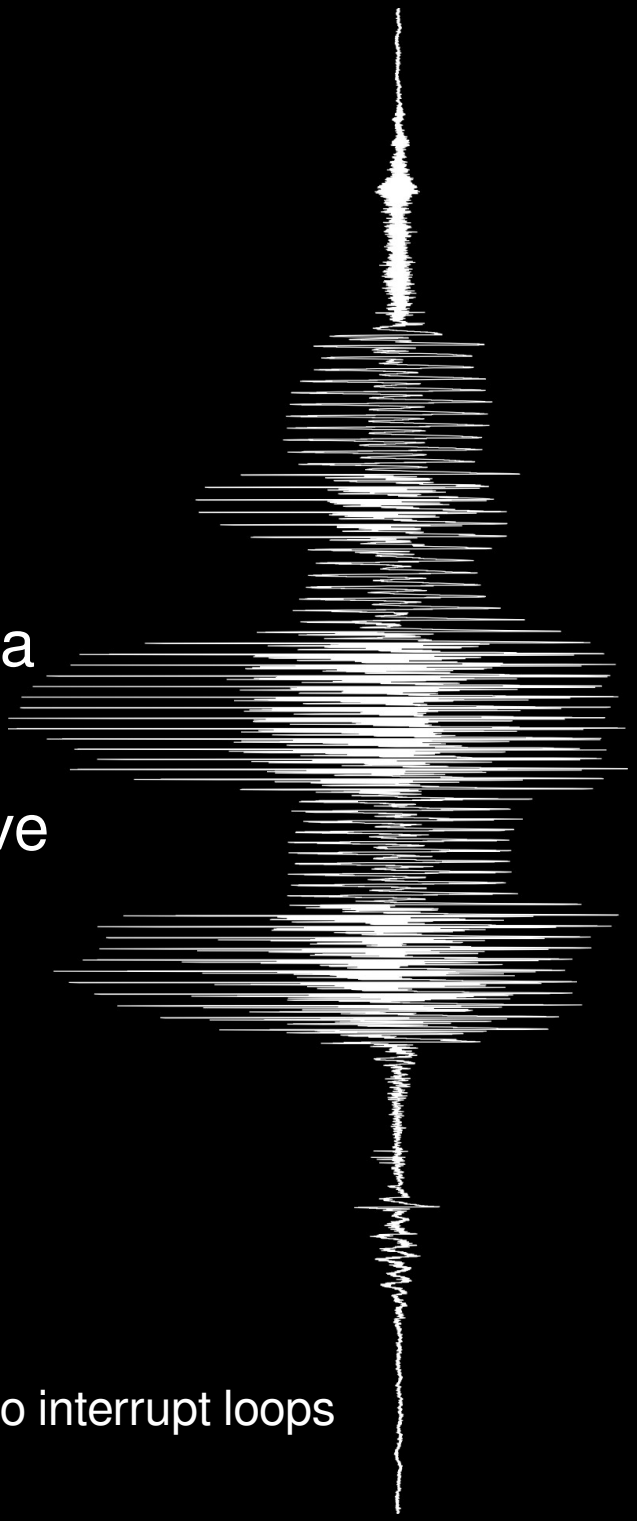
upon quitting

- `exit [TEXT]`

  ends the execution* of the script and prints a message (if provided)

  this directive works different to the rest we've seen: everything that follows it until the end of the line is the string

  this is a remnant of the old syntax



* this makes useful as a quick and dirty way to to interrupt loops

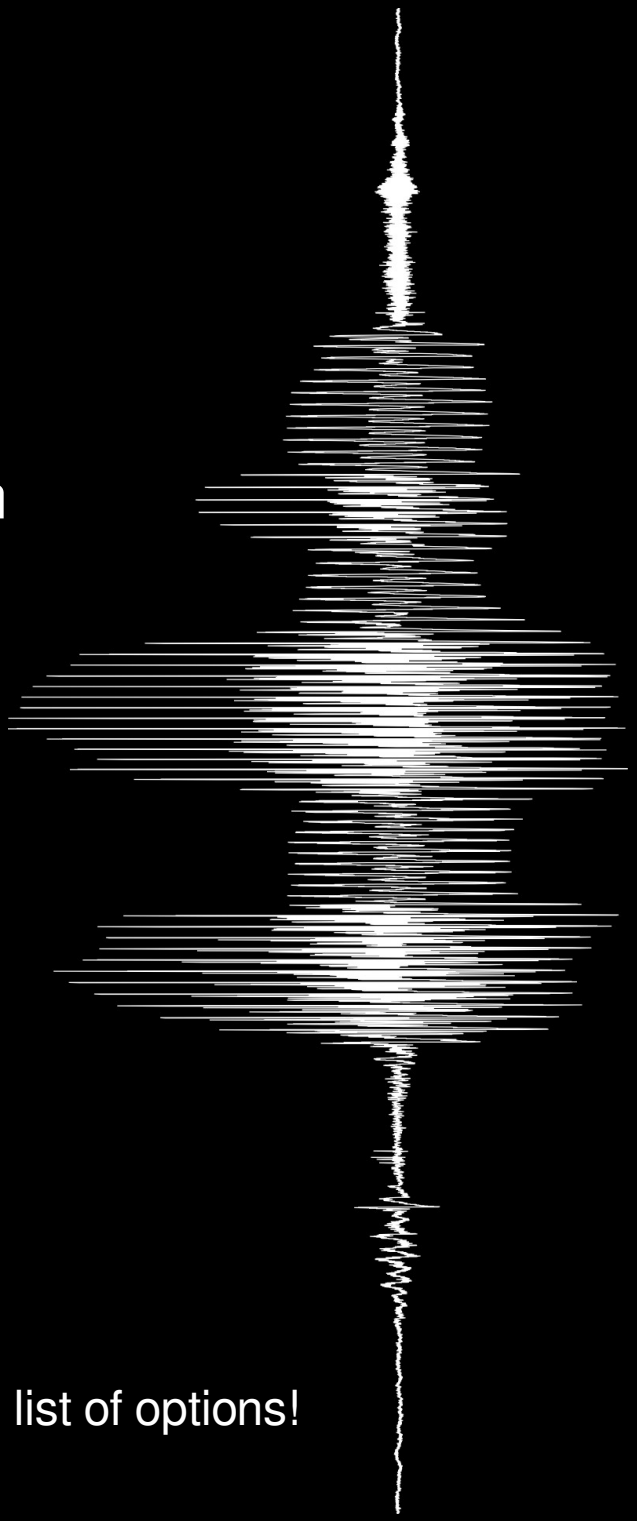
before or during execution

- `form … endform`

  `form` allows us to create a dialog box which will come up upon execution for user input

```
form Title...
   natural positiveInteger defaultValue
   real realNumber defaultValue
   ...
   word stringNoSpaces defaultValue
   text string defaultValue
   choice multipleChoice defaultValue
      button Text
      button Text
   comment Text
   etc...
endform
```

check the manual for the full list of options!

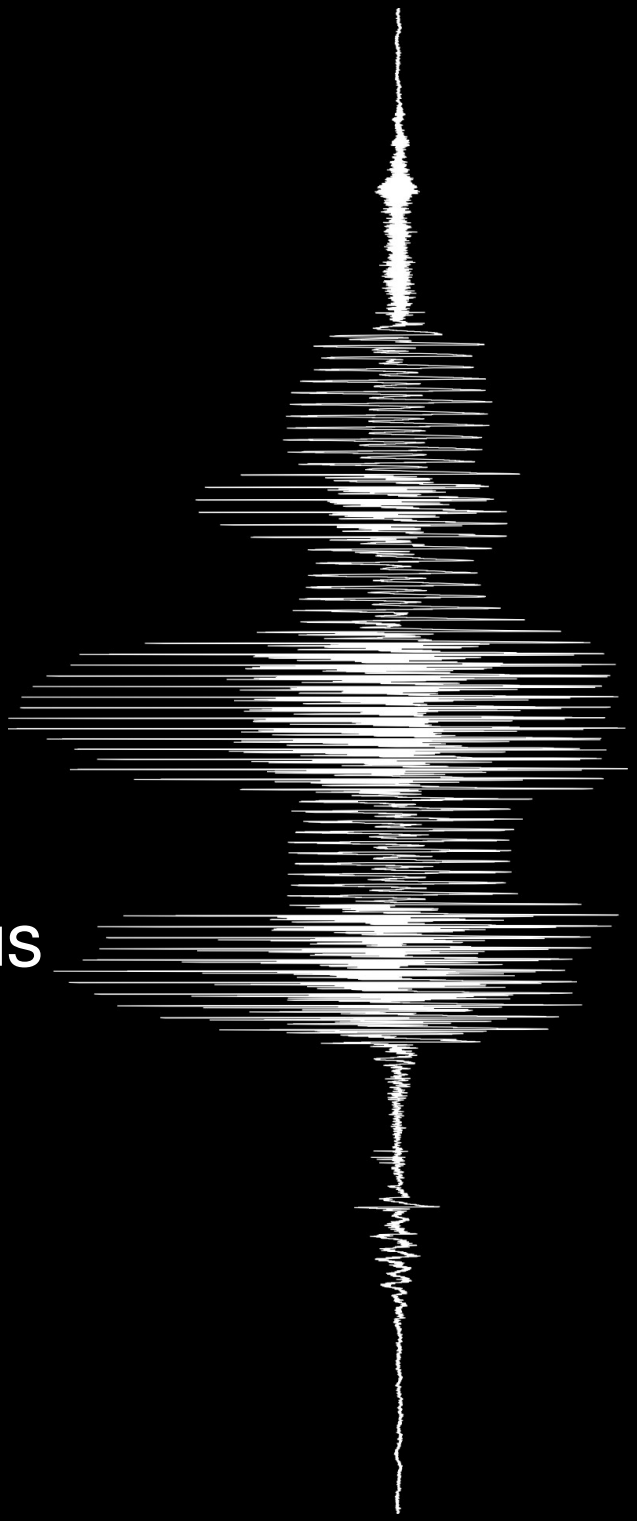


interacting with the file system

- since they are buttons, commands like
  - `Read from file...`
  - `Open long sound file...`
  - `Save as text file...`

  are also available from scripts

  but praat allows for some more ambitious modifications with the file system



interacting with the file system

- `fileReadable(a$)`

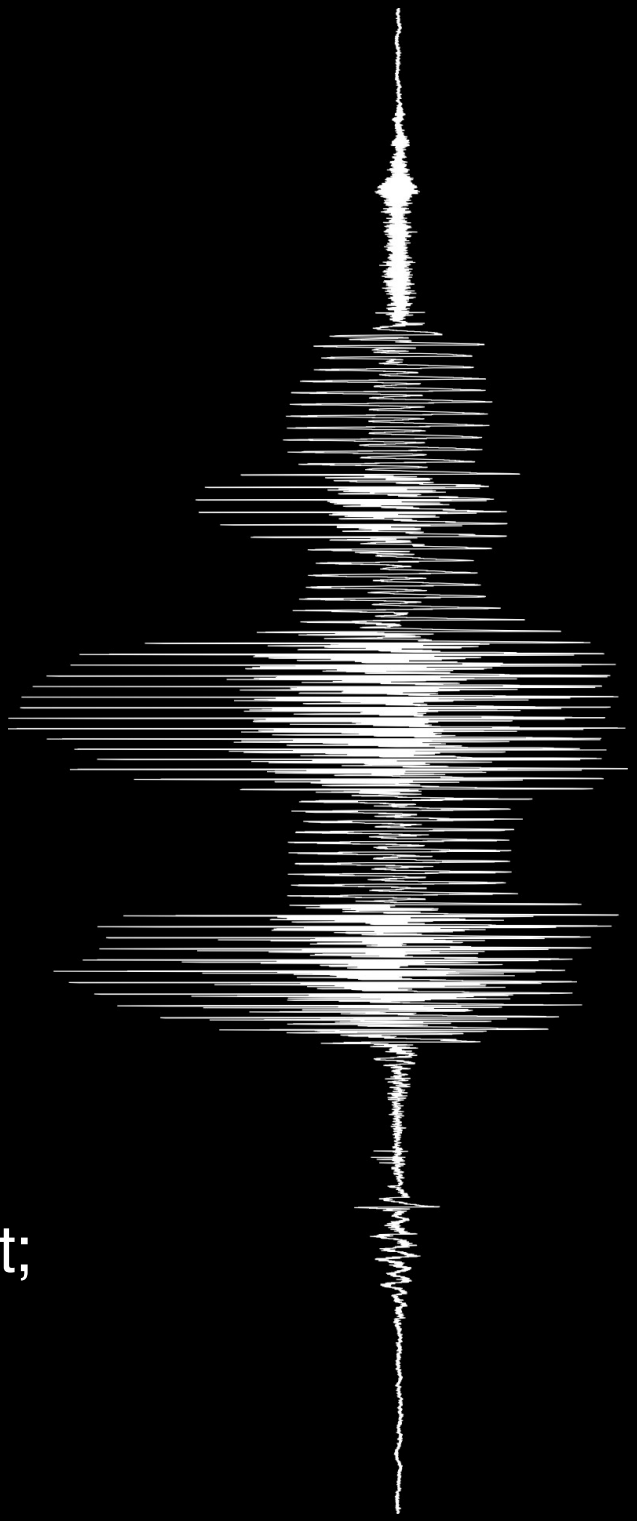
  returns 1 if file `a$` exists and is readable (uses relative paths)

- `createDirectory(a$)`

  attempts to create the directory specified in `a$`; returns 1 if possible (even if it already existed), 0 if unable to (because of permissions, for example)

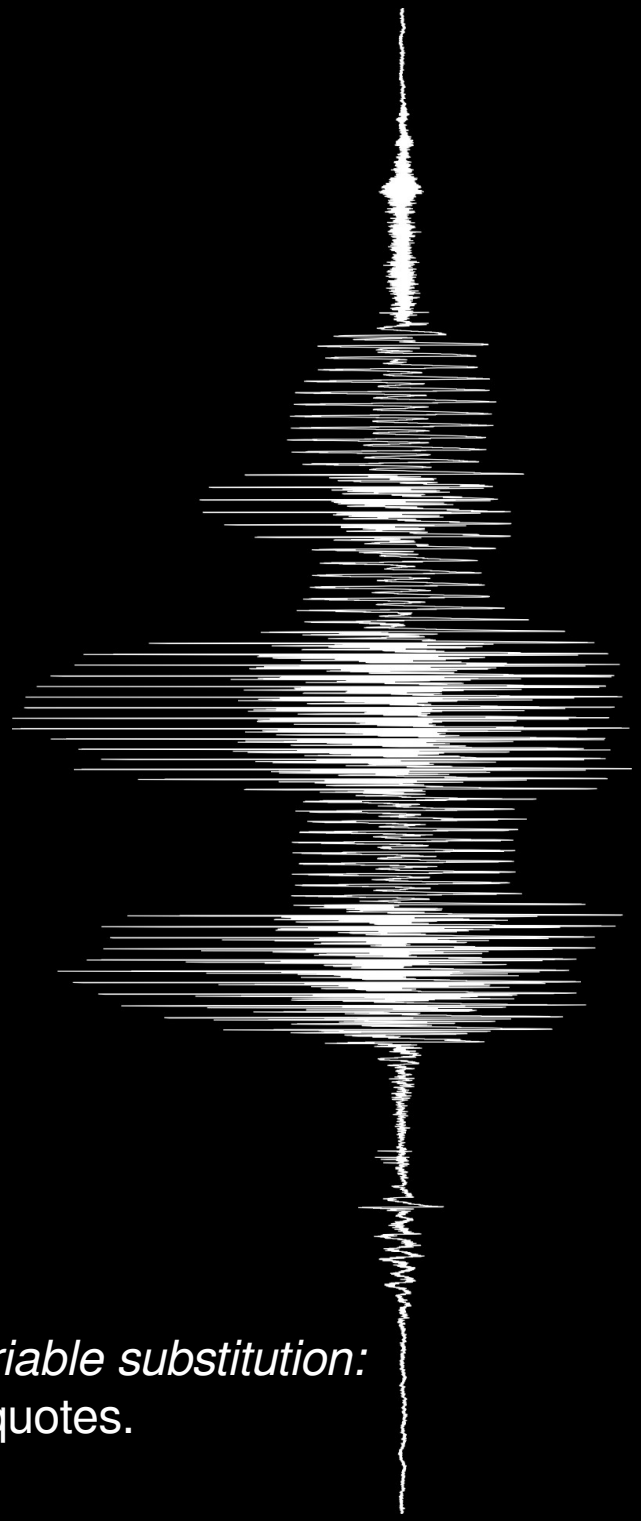
- `deleteFile(a$)`

  always returns 1. if file `a$` exists, it deletes it; if not, it does nothing



interacting with the file system

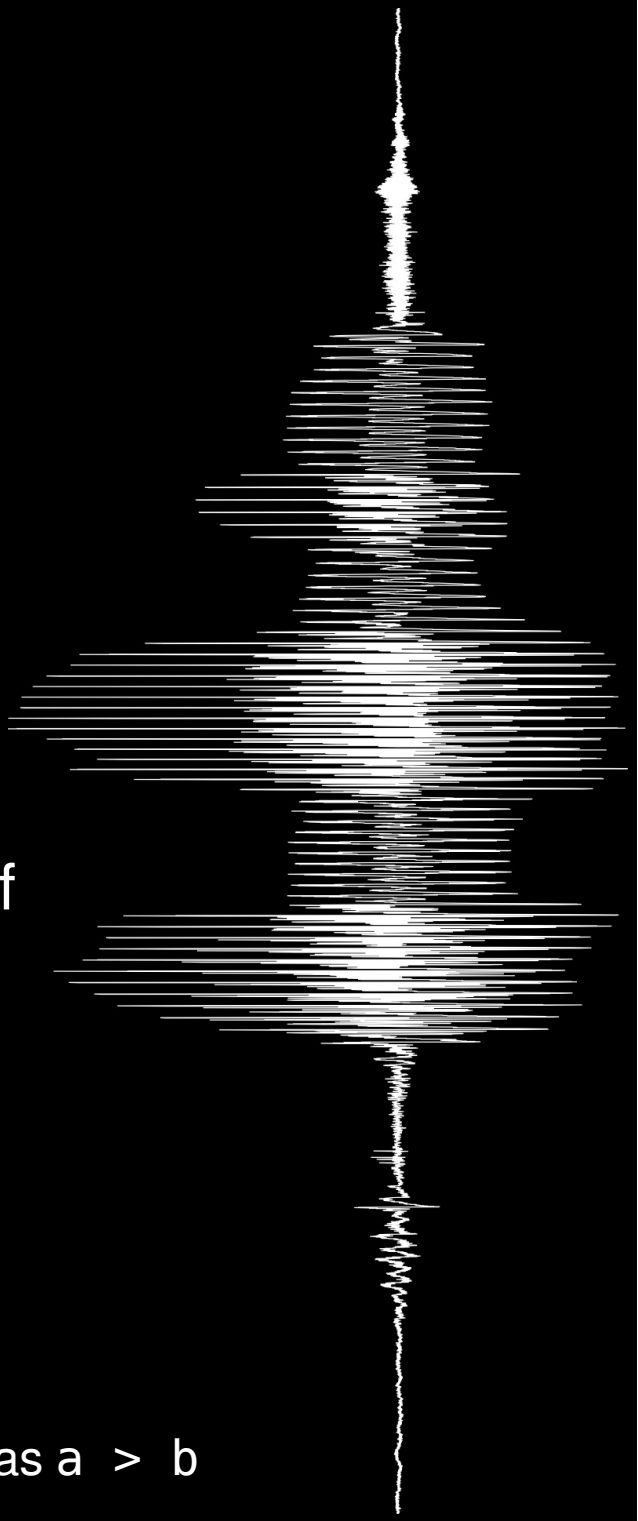
```
replace = 0
name$ = selected$ ("Sound")
filename$ = name$ + ".wav"
file_exists = fileReadable(filename$)
if file_exists = 1 and replace = 1
   deleteFile(filename$)
elsif file_exists = 1 and replace = 0
   exit File 'replace$' exists. Aborting...
endif
do("Save as WAV file...", filename$)
appendInfoLine(filename$, " has been saved")
```



those single quotes around `'replace$'` cause *variable substitution:*
it will be replaced by the contents of the variable in quotes.
once again, those are remnants of the old syntax

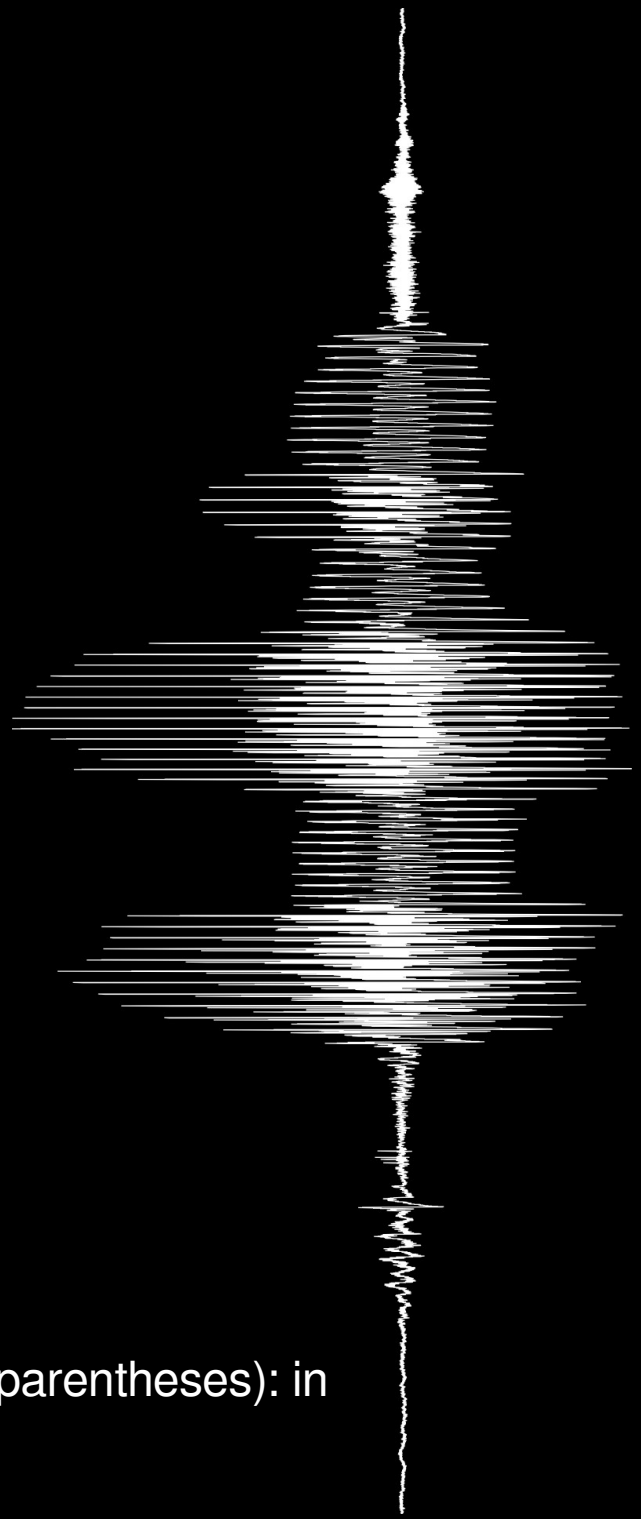
interacting with the file system

- `a$ > b$`
  - saves the contents of `a$` in file `b$`
- `a$ >> b$`
  - appends the contents of file `a$` to the end of file `b$`, or, if `b$` does not exist, creates it
- `a$ < b$`
  - saves the contents of `b$` in `a$`



remember! `a$ > b$` is not the same as `a > b`

interacting with the file system

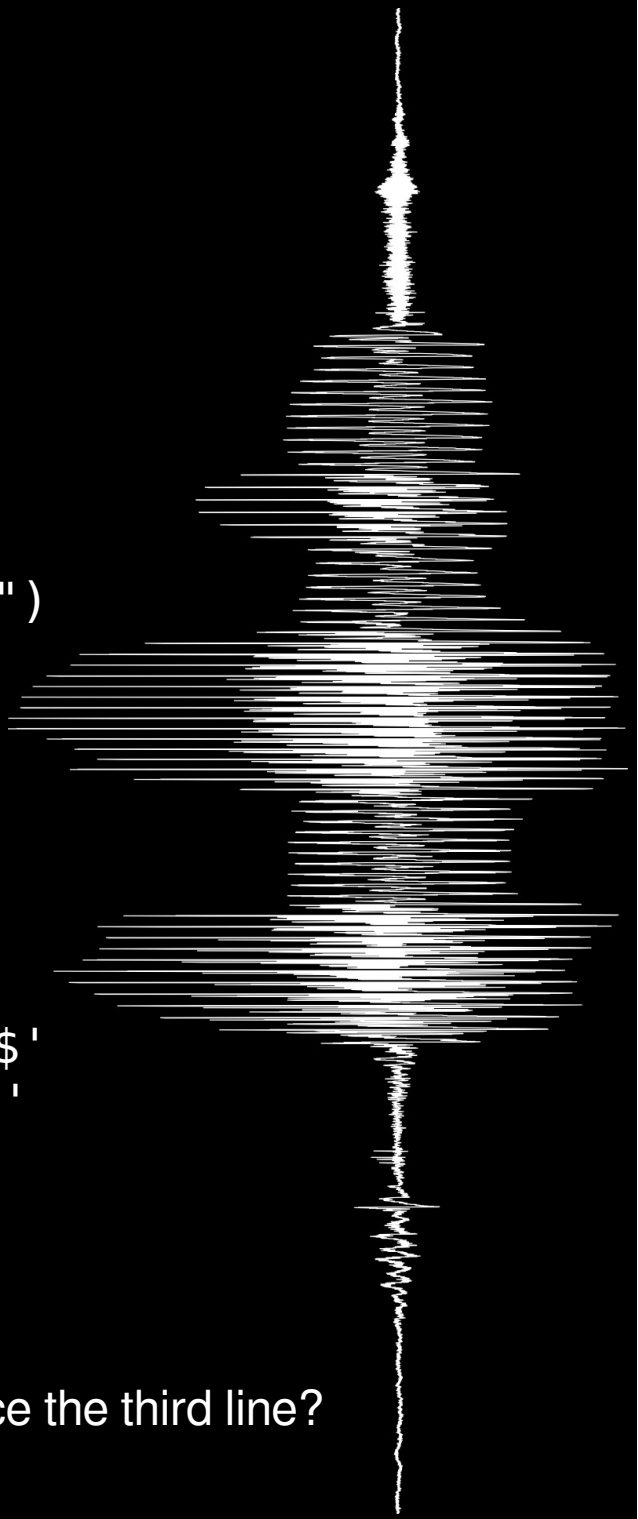


- `fileappend a$ b$`
  - appends the contents of `b$` to `a$`
- `filedelete b$`
  - deletes file `b$` if it exists

these are not functions (note the lack of parentheses): in the manual they are called *directives*

interacting with the file system

```
config_file$ = "config.ini"
a = fileReadable(config_file$)
if a
   config$ < 'config_file$'
   height = extractNumber(config$, "Height = ")
   width = extractNumber(config$, "Width = ")
   name$ = extractLine$(config$, "Name = ")
else
   example$ = "Height = 0'newline$'" +
      ... "Width = 0'newline$'" +
      ... "Name = Ferdinand"
   example$ > 'config_file$'
   exit 'config_file$' does not exist'newline$'
      ...an example has been created'newline$'
endif
appendInfoLine("Height=", height, tab$,
   ..."Width=", width, tab$,
   ..."Name=", name$)
```



did you notice the third line?

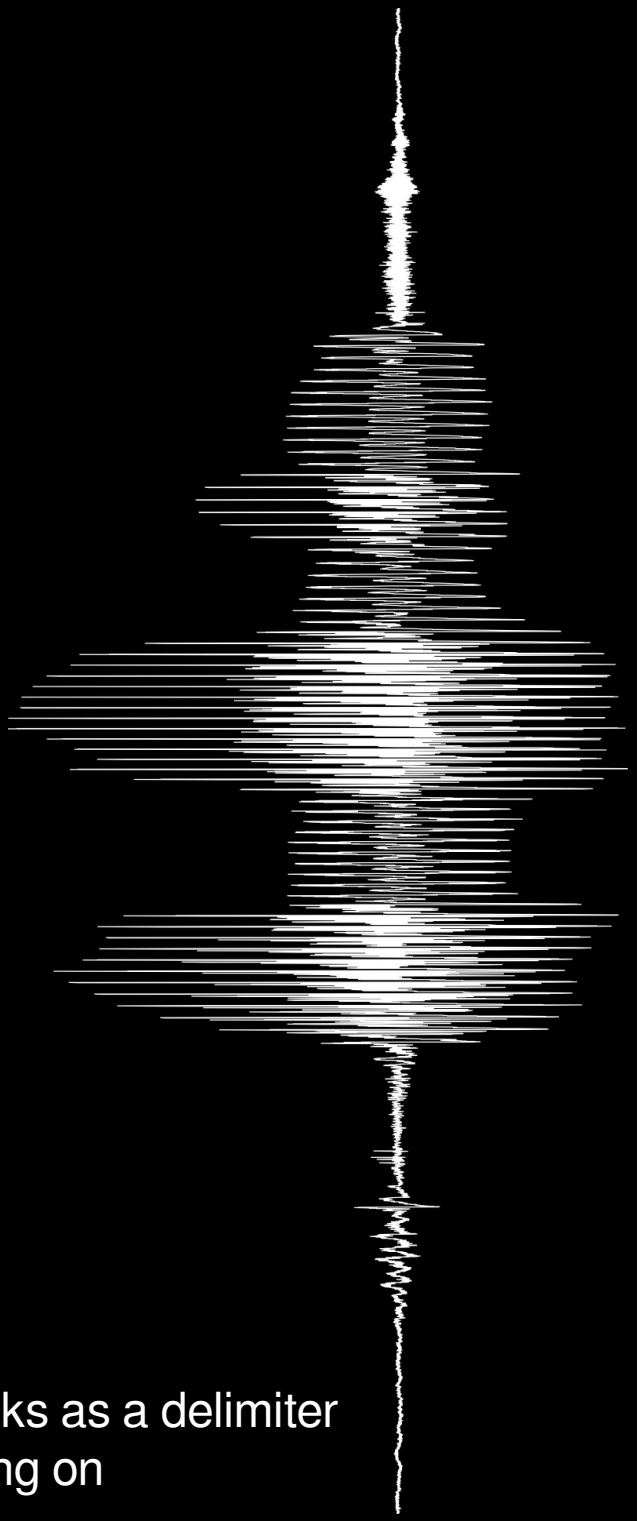
the last functions

- `chooseReadFile$(a$)`
- `chooseWriteFile$(a$, b$)`

  open dialog boxes with title `a$` in which the user can select a file to read or to write; in this last case, `b$` is a suggested file name

- `chooseDirectory$(a$)`

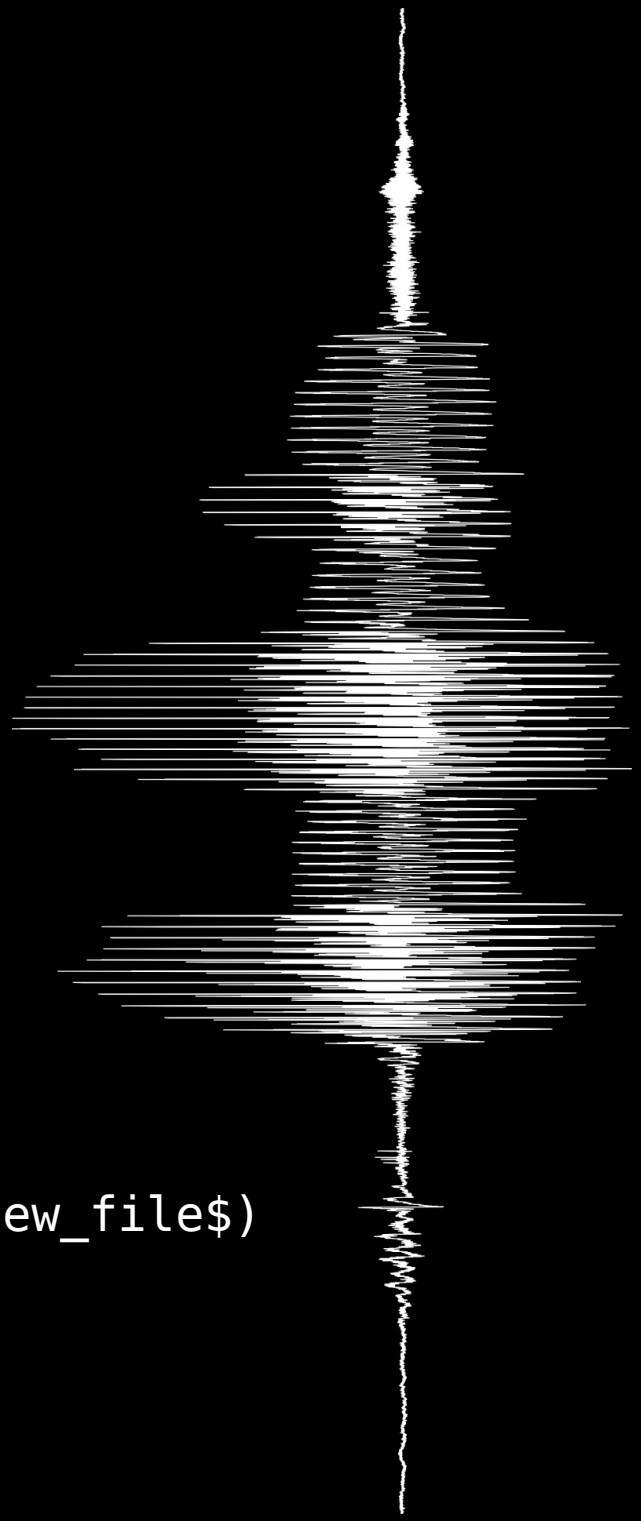
  opens a dialog box for selecting a directory from the directory structure, with title `a$`



in praat, a forward slash (/) works as a delimiter no matter the system it is running on

the last functions

```
filename$ = chooseReadFile$("Open sound...")
if filename$ = ""
   exit
endif
strlen = length(filename$)
dot = rindex(filename$, ".")
new_file$ = left$(filename$, dot-1)
   ...+ "_reversed"
   ...+ right$(filename$, (strlen-dot)+1)
if !fileReadable(filename$)
   do("Read from file...", filename$)
else
   exit Could not read 'filename$'
endif
do("Reverse")
savefile$ = chooseWriteFile$("Save as...", new_file$)
if savefile$ != ""
   do("Save as WAV file...", savefile$)
endif
```



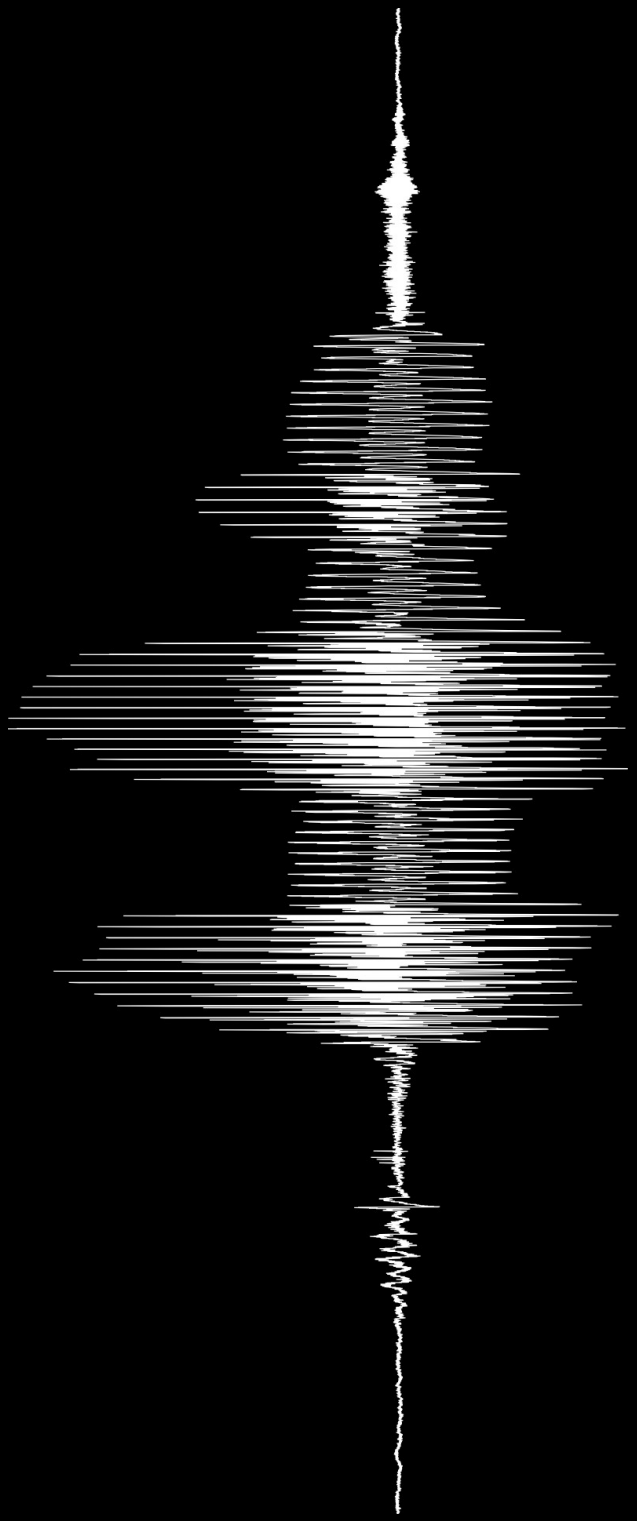
other resources

the example scripts shown here (as well as the latest version of this presentation) are available at

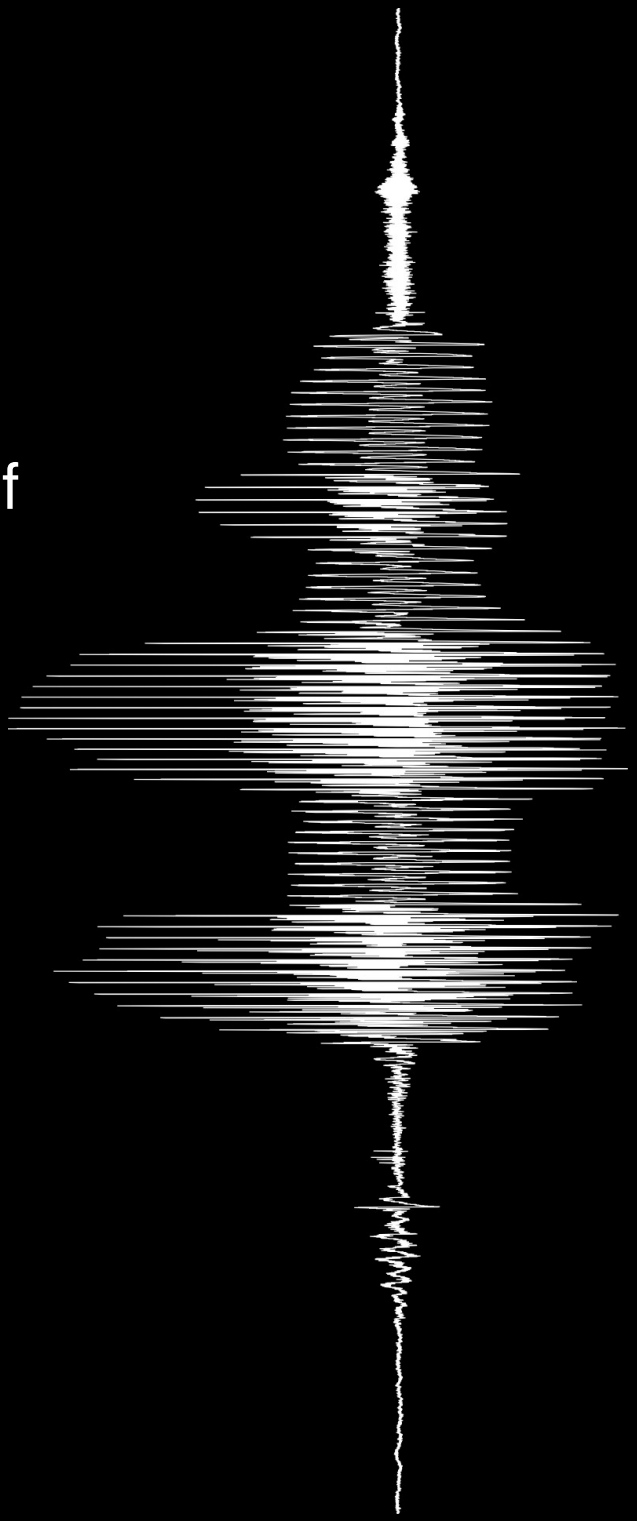
http://www.pinguinorodriguez.cl/tutorials/praatscript.html

don't forget to check the praat manual;
it has loads of information

Praat Manual - Scripting



other resources



you can (and should!) also check the mailing list of praat users, where there are often interesting tips and suggestions, as well as questions with their answers.

and if you have the chance to give back to the community, all the better!

http://uk.groups.yahoo.com/group/praat-users/

you can send emails to the list to

praat-users@yahoogroups.co.uk

other resources

and of course, look online for the many websites that aggregate scripts and tutorials. here's a few ideas:

University of Iowa - Praat Script Archives

UCLA Praat script resources

and Scott Sadowsy has developed a syntax highlighting file for praat to be used with Notepad++, freely available at

http://www.sadowsky.cl/praat.html