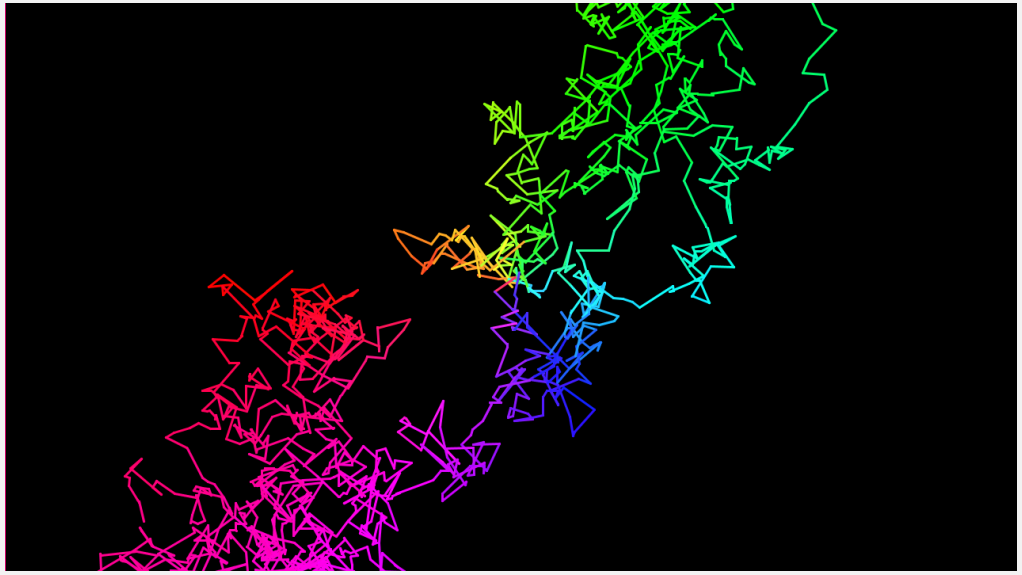
POLITECNICO
MILANO 1863

IMAGE AND SOUND
ISPG
PROCESSING GROUP

CREATIVE PROGRAMMING AND COMPUTING

Lab: Grammars and server-side programming

LEVY FLIGHT SONIFICATION

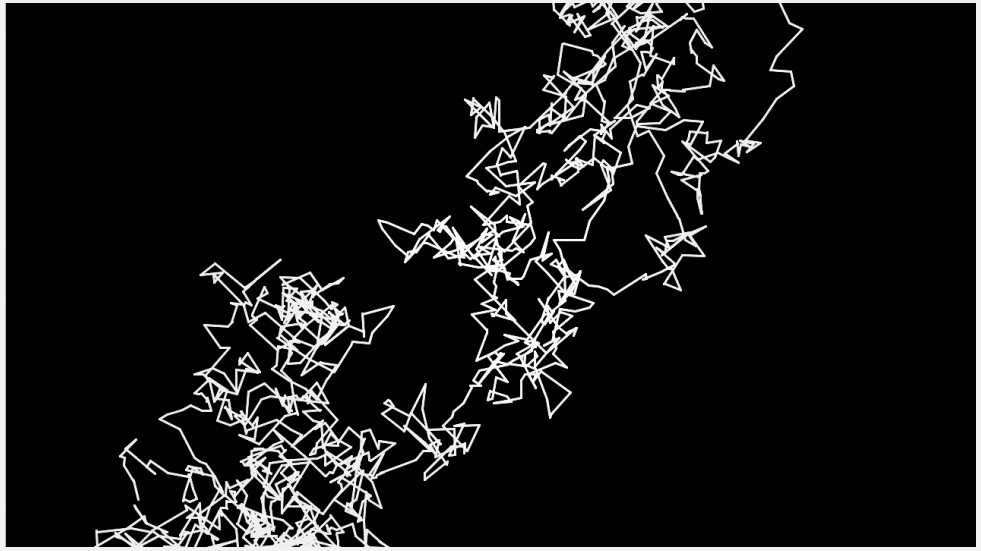


LEVY FLIGHT SONIFICATION

Levy Flight is an algorithm of random walk that combines rule-based behavior with creativity brought by randomness and probability distribution.

The agent Walker does the following steps

1. Start from a Position
2. Choose a random direction where to move towards
3. Choose a stepsize using a *MonteCarlo* sample
4. Move toward direction * stepsize * constant scale
5. Draw lines of the movement



LEVY FLIGHT SONIFICATION

MonteCarlo sampling

Two-step sampling

1. $R_1 = random(1), p = random(1);$
2. $R_2 = random(1);$
3. $R_2 < p$ → return R_1
4. $R_2 \geq p$ → back to step 1

Compute R_1 as a candidate stepsize, then probability p that R_1 is a candidate and R_2 adversary likelihood.

Pick R_1 when $p > R_2$

GRAMMAR-BASED COMPOSITION

Ex 1: implement methods `Walker.update`, `Walker.draw` and function `float montecarlo`

```
# ex1.pde
Walker walker; PVector CENTER_SCREEN;
float ALPHA_BACKGROUND=0;
void setup() {
  size(1280,720);
  walker= new Walker();
  CENTER_SCREEN=new PVector(width/2,
                            height/2);
  background(0);
}
void draw() {
  fill(0, ALPHA_BACKGROUND);
  strokeWeight(0);
  rect(0,0,width, height);
  walker.update();
  walker.draw();
}
```

```
# Walker.pde
class Walker {
  PVector position, prevPosition;
  Walker() {
    this.position=CENTER_SCREEN.copy();
    this.prevPosition=CENTER_SCREEN.copy();
  }
  void draw(){stroke(255); // your code }
  void update() {
    PVector step;
    // your code: use montecarlo sampling
    this.prevPosition=this.position.copy();
    this.position.add(step);
  }
}
float montecarlo() {
  while (true) {/* your code */}}
```

LEVY FLIGHT SONIFICATION

MonteCarlo sampling

Two-step sampling

1. $R_1 = random(1), p = random(1);$
2. $R_2 = random(1);$
3. $R_2 < p$ → return R_1
4. $R_2 \geq p$ → back to step 2

One-step sampling

1. $R_1 = random(1), p = F(R_1);$
2. $R_2 = random(1);$
3. $R_2 < p$ → return R_1
4. $R_2 \geq p$ → back to step 2

LEVY FLIGHT SONIFICATION

MonteCarlo sampling

Two-step sampling

1. $R_1 = random(1), p = random(1);$
2. $R_2 = random(1);$
3. $R_2 < p$ → return R_1
4. $R_2 \geq p$ → back to step 2

One-step sampling

1. $R_1 = random(1), p = F(R_1);$
2. $R_2 = random(1);$
3. $R_2 < p$ → return R_1
4. $R_2 \geq p$ → back to step 2

The probability that R_1 is a good candidate is computed as a function of R_1 itself: $p = F(R_1)$

- Shape the function accordingly to the desired stepsize R_1
- p gets higher when R_1 is similar to the desired one
- what if we want small steps to be more likely?

LEVY FLIGHT SONIFICATION

MonteCarlo sampling

Two-step sampling

1. $R_1 = random(1), p = random(1);$
2. $R_2 = random(1);$
3. $R_2 < p \rightarrow$ return R_1
4. $R_2 \geq p \rightarrow$ back to step 2

One-step sampling

1. $R_1 = random(1), p = F(R_1);$
2. $R_2 = random(1);$
3. $R_2 < p \rightarrow$ return R_1
4. $R_2 \geq p \rightarrow$ back to step 2

The probability that R_1 is a good candidate is computed as a function of R_1 itself: $p = F(R_1)$

- Shape the function accordingly to the desired stepsize R_1
- p gets higher when R_1 is similar to the desired one
- what if we want small steps to be more likely?

$F(R_1) = (1 - R_1)^k$ with $k \geq 1$

GRAMMAR-BASED COMPOSITION

Ex 2: implement the one-step MonteCarlo

```
# ex2.pde
int MONTECARLO_STEPS=1;
PVector CENTER_SCREEN;
# ...
```

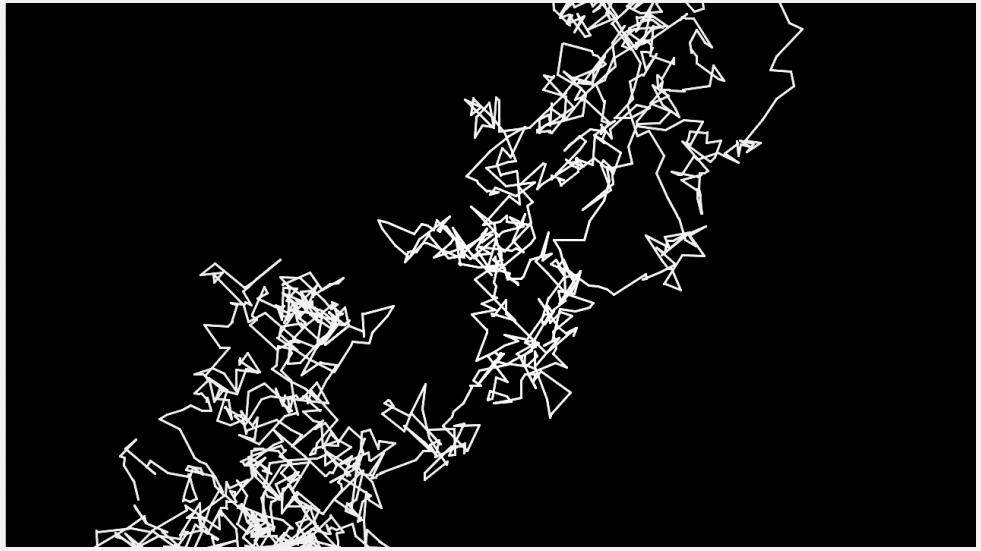
```
# Walker.pde
# ...
float montecarlo() {
  while (true) {
    if(MONTECARLO_STEPS==2){
      // as before
    }
    if(MONTECARLO_STEPS==1){
      // your code
    }
  }
}
```

LEVY FLIGHT SONIFICATION

Let's add one more step

The agent Walker does the following steps

1. Start from a Position
2. Choose a random direction where to move towards
3. Choose a stepsize using a *MonteCarlo* sample
4. Move toward direction * stepsize * constant scale
5. Keep drawing the line

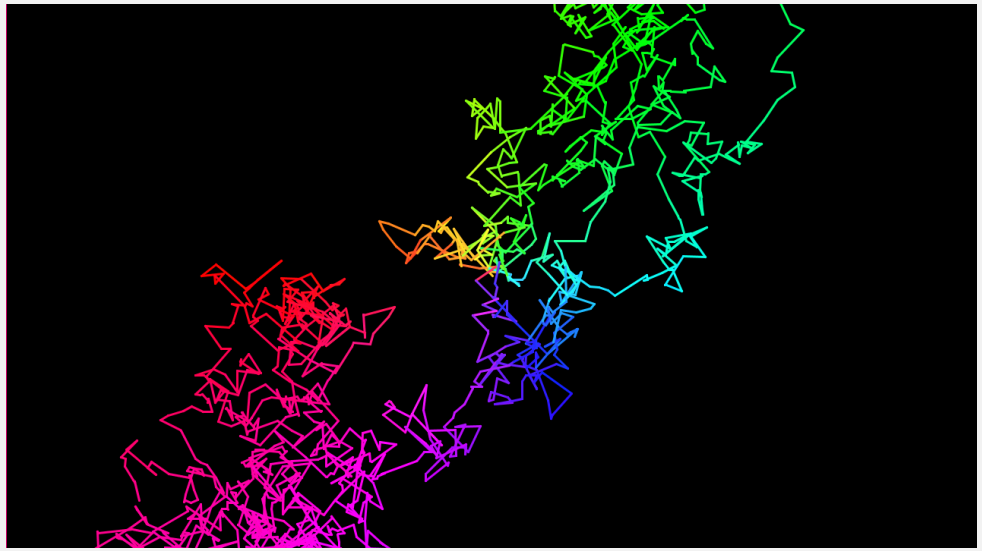


LEVY FLIGHT SONIFICATION

Let's add one more step

The agent Walker does the following steps

1. Start from a Position
2. Choose a random direction where to move towards
3. Choose a stepsize using a *MonteCarlo* sample
4. Move toward direction * stepsize * constant scale
5. Keep drawing the line
6. Changes color, shape, stroke weight
   following walker's parameters



GRAMMAR-BASED COMPOSITION

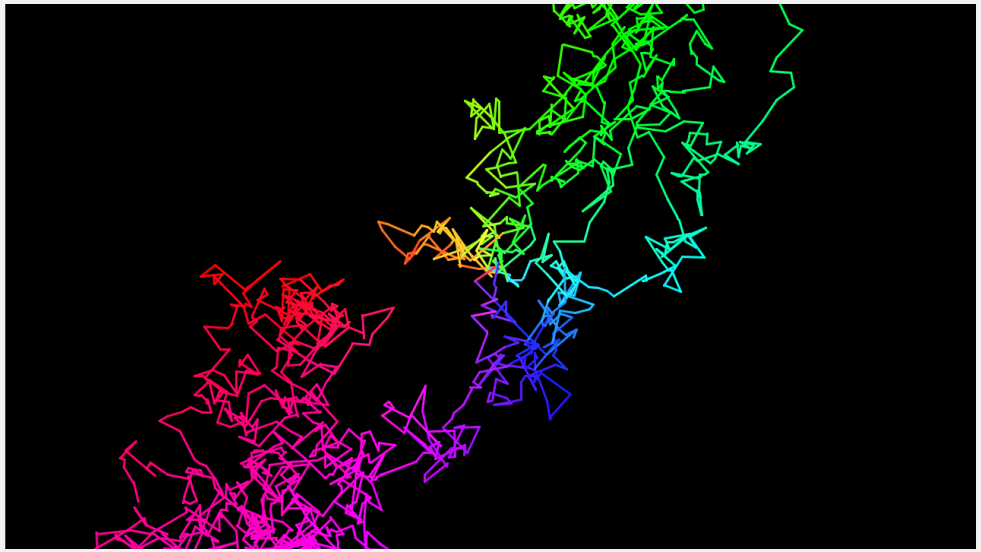
Ex 3: change `Walker.draw` so it changes stroke color depending on the walk.
For example:

- Map the x and y coordinates of the position into two color channels, leaving the other as a fixed value
- Map the length of the step, or the distance from the center of the screen into a brightness value
- Map the angle of the new position with respect to the center of the screen into a hue value.
  - You can use as `PVector.heading()` as shown below

```
# Walker.pde
class Walker {
  #...
  void draw(){
    Color c;
    /* your code here */
    stroke(c);
    /* your code here*/
  }
}# ...
```

```
Pvector fromCenter=PVector.sub(this.position, CENTER_SCREEN);
float normAngle= map(fromCenter.heading(), -PI, PI, 0, 1);
```

LEVY FLIGHT SONIFICATION

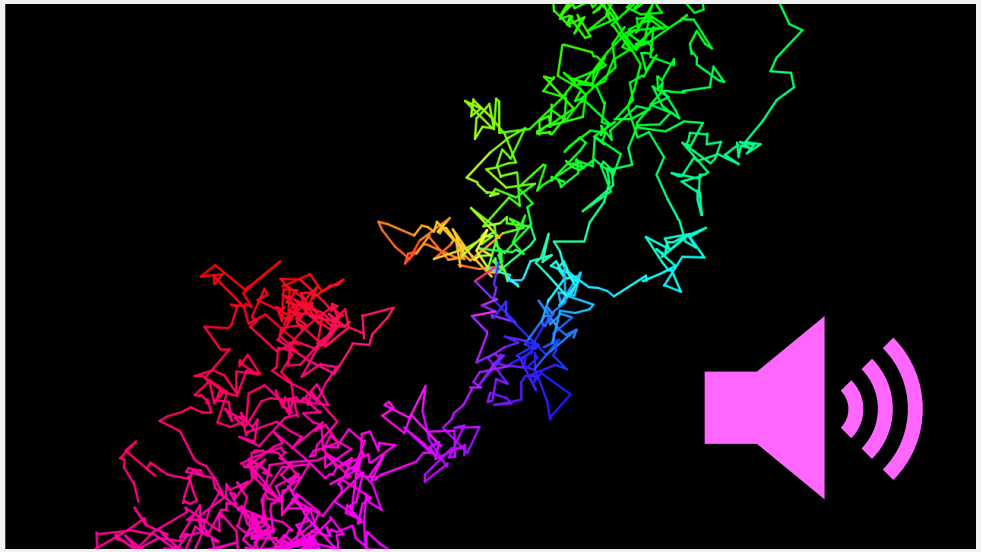


Let's add one more step

The agent Walker does the following steps

1. Start from a Position
2. Choose a random direction where to move towards
3. Choose a stepsize using a *MonteCarlo* sample
4. Move toward direction * stepsize * constant scale
5. Keep drawing the line
6. Changes color, shape, stroke weight
   following walker's parameters

LEVY FLIGHT SONIFICATION



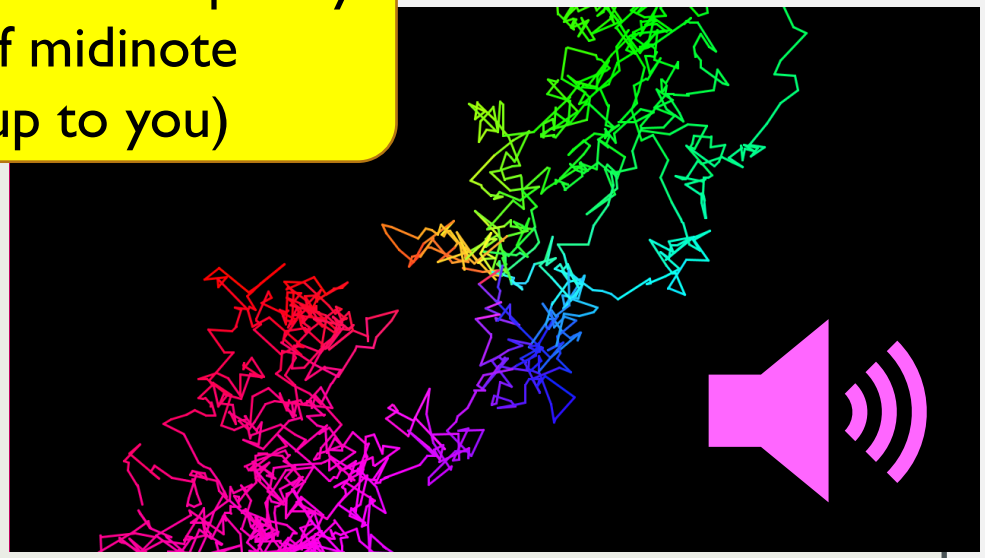
Let's add one more step

The agent Walker does the following steps

1. Start from a Position
2. Choose a random direction where to move towards
3. Choose a stepsize using a *MonteCarlo* sample
4. Move toward direction * stepsize * constant scale
5. Keep drawing the line
6. Changes color, shape, stroke weight
   following walker's parameters
7. Play our moog instrument
   following walker's parameters

LEVY FLIGHT SONIFICATION

We set continuous frequency instead of midinote (but it's up to you)



```
# moog.scd
(SynthDef("moog", {
    arg freq=60, vibr=0, cutoff=0.5, amp=0;
    var osc1, osc2, osc3, f0, vib_int;
    var cutoff_freq, fil_osc;
    f0=exp(vibr*0.035)*freq;
    // etc...
)
// Use this command to start the server
NetAddr("127.0.0.1",57120);

OSCdef('OSCreceiver',{
    arg msg; var freq, amp, cutoff, vibr;
    freq=msg[1]; amp=msg[2]; cutoff=msg[3]; vibr=msg[4];
    ~instr.set(\freq,freq, \amp,amp, \cutoff, cutoff, \vibr, vibr);
    }, "/note_effect",);
)
```

We send all the parameters in the same message

GRAMMAR-BASED COMPOSITION

Ex 4: connect the walk to the moog: implement `Walker.computeEffect()`

For example:

- Map the x and y coordinates of the position into amp and freq
- Map the sum of x and y coordinates to frequency
- Map the x coordinate to the cutoff factor
- Map the vibrato to the step
  - Positive if the Levy Flight is going further from the center
  - Negative if the Levy Flight is going toward the center

```
# Walker.pde
class Walker { //...
  float freq, amp;
  float cutoff, vibrato;
  Walker() { //initialize...}
  //...
  void computeEffect(){
    this.freq=0; this.amp = 0;
    this.cutoff=0;
    this.vibrato=0;
    // your code
  }
}
```

GRAMMAR-BASED COMPOSITION

Ex 4: connect the walk to the moog: implement `Walker.computeEffect()`

```
# ex4.pde
import oscP5.*; import netP5.*;
OscP5 oscP5; NetAddress ip_port;
void setup() {//...
  oscP5 = // my code...
  ip_port = // my code...}

void sendEffect(Walker w){
  OscMessage msg = // my code...
  msg.add(w.freq); msg.add(w.amp);
  msg.add(w.cutoff); msg.add(w.vibrato);
  oscP5.send(msg, ip_port);
}

void draw() { //...
  walker.computeEffect();
  sendEffect(walker);
}
```

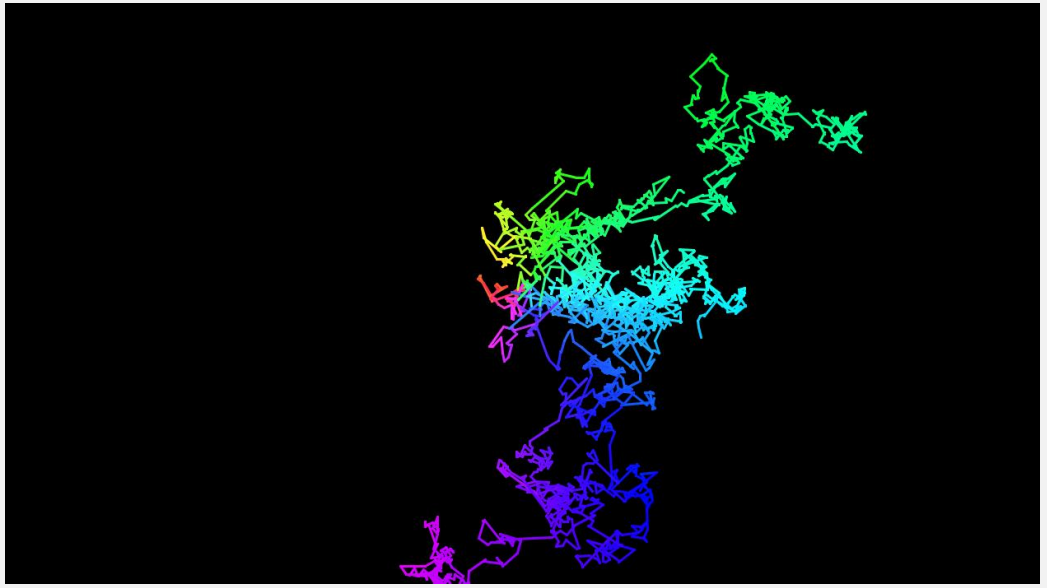
```
# Walker.pde
class Walker { //...
  float freq, amp;
  float cutoff, vibrato;
  Walker() { //initialize...}
  //...
  void computeEffect(){
    this.freq=0; this.amp = 0;
    this.cutoff=0;
    this.vibrato=0;
    // your code
  }
}
```

LEVY FLIGHT SONIFICATION

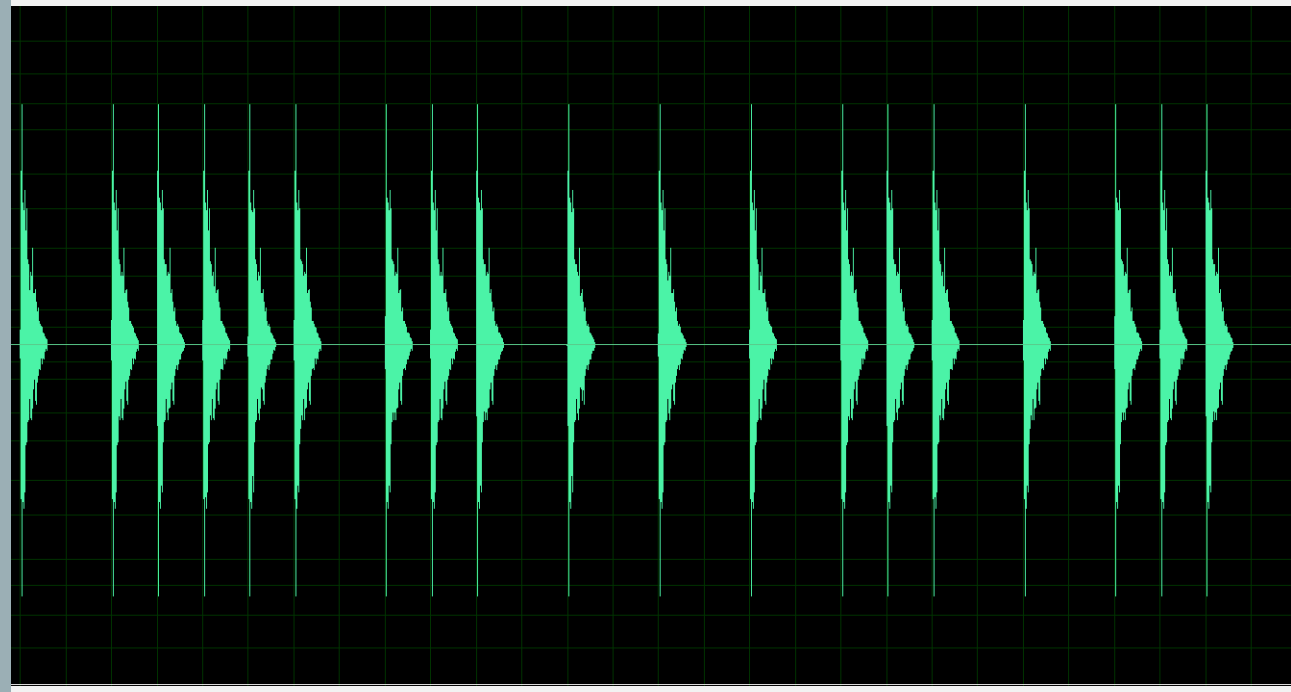
Levy Flight is an algorithm of random walk that combines rule-based behavior with creativity brought by randomness and probability distribution.

- You can use them to draw a random walk
- You can use it as a source to control animation or sounds
- You can do the opposite: control the step of the random walk with some meaningful parameter

- Also: from single-agent to multi-agent



GRAMMAR-BASED COMPOSITION



GRAMMAR-BASED COMPOSITION

- Grammars are composed of *elements* and *rules* to use them
- A formal grammar is a set of rules to expand high-level symbols (**non-terminal symbols** or **sentences**) into more detailed sequences of symbols (**terminal symbols** or **words**) representing elements of formal languages

- A music score follows a grammar, as for it is composed by elements and rules
  - Elements are notes and pauses, with pitch and duration
  - Rules are how many notes you can fit in a measure, how you define a measure, how you read them

GRAMMAR-BASED COMPOSITION

- We will start from the grammar defined by Keller and Morrison in 2007 in the creation of the Impro-visor
  - Educational grammar-based software for automatic Jazz solo melody creation
  - *"Impro-Visor was designed to assist the soloist, who may not yet have a strong theory background, in the process of creating solos"*

- We will focus on their grammar for rhythmic sequences as they can be created without the need of modelling of melodies and harmony

GRAMMAR-BASED COMPOSITION

Create a grammar-based composition

1. Define a grammar as a set of elements and rules
2. Define a starting sentence
3. Keep replacing the sentences (non-terminal symbols) following the rules until only words (terminal symbols) are present
4. Replace words with the appropriate notes

GRAMMAR-BASED COMPOSITION

Define a grammar as a set of elements and rules

Example with a basic grammar based on 4/4 signature 𝐂

Terminal symbols (words)

- **h**: half-note, i.e., half measure
- **q**: quarter-note 

Non-terminal symbols (sentences)

- **S** → M, SM
- **M** → HH
- **H** → h, qq

GRAMMAR-BASED COMPOSITION

Define a grammar as a set of elements and rules

Example with a basic grammar based on 4/4 signature 𝄴

Terminal symbols (words)

- **h**: half-note, i.e., half measure 𝅗𝅥
- **q**: quarter-note ♩

Non-terminal symbols (sentences)

- **S** → M, SM
- **M** → HH
- **H** → h, qq

a Sequence can become one measure or one measure followed by a sequence

a Measure is composed by two halves

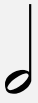
Half a measure can be composed of a half-note or two quarter-notes

GRAMMAR-BASED COMPOSITION

Define a grammar as a set of elements and rules

Example with a basic grammar based on 4/4 tempo

Terminal symbols (words)

- **h**: half-note, i.e., half measure 
- **q**: quarter-note 

Non-terminal symbols (sentences)

- **S** → M, SM
- **M** → HH
- **H** → h, qq

GRAMMAR-BASED COMPOSITION

Define a starting sentence

- A common starting sentence is **S**
- We can decide to set a fixed number of measures for our composition
- Let's create a loop of four measures **MMMM**

GRAMMAR-BASED COMPOSITION

S → M, SM
M → HH
H → h, qq

Keep replacing the sentences until only words are present

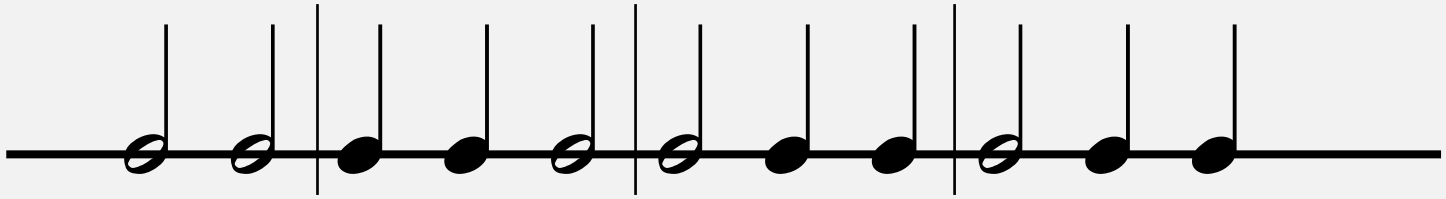
In this stage we will randomly pick one sentence at time and then replace with a random (terminal or non-terminal) symbol, following the rules. For example:

- **MMMM** → **MHHMM** → **HHHHMM** → **HHqqHMM** → h**H**qq**HM**M→
- h**H**qq**HMHH** → h**h**qq**HMHH** → hhqq**HMH**qq → hhqq**HM**hqq →
- hhqq**HHH**hqq → hhqq**HH**qqhqq → hhqqh**H**qqhqq →
- hhqqhhqqhqq

GRAMMAR-BASED COMPOSITION

Replace words with the appropriate notes

- hhqqhhqqhqq



Bonus: generate a track with the notes placed like this

GRAMMAR-BASED COMPOSITION

Let's create a music composition based on grammars in Python

- The sequence will be a string
  - Non-terminal symbols in UPPER case
  - We start with 8 measures as a start sequence
- The grammar is a dictionary
  - Maps each non-terminal symbol to the corresponding terminal and non-terminal symbols
- We initialize a class `Grammar_Sequence` with the desired grammar
  - `create_sequence` takes in input the `start_sequence` and digest it to create a sequence of terminal symbols `G.sequence`

```
# main.py
NUM_M=8
START_SEQUENCE="M"*NUM_M
ss=START_SEQUENCE
basic_grammar={
    "S":["M", "SM"],
    "M": ["HH"],
    "H": ["h", "qq"]
}
b_g=basic_grammar
G=Grammar_Sequence(b_g)
G.create_sequence(ss)
```

GRAMMAR-BASED COMPOSITION

Let's create a music composition based on grammars in Python

- Then we use a class `Composer` initialized with a sample sound and the desired samplerate
- We convert the sequence from string to a signal using `C.create_sequence`
- We write the created sequence as an output wavfile

We need a mapping between words and note duration

```
# ex1.py
NUM_M=8
START_SEQUENCE="M"*NUM_M
ss=START_SEQUENCE
basic_grammar={
    "S":["M", "SM"],
    "M": ["HH"],
    "H": ["h", "qq"]
}
b_g=basic_grammar
G=Grammar_Sequence(b_g)
G.create_sequence(ss)
C=Composer("sound.wav", sr=SR)
C.create_sequence(G.sequence)
C.write("out.wav")
```

GRAMMAR-BASED COMPOSITION

We need a mapping between words and note duration

and a function to convert from a sequence of words to a list of corresponding durs

Let's start playing with some grammars

```
# classes.py
default_word_dur={# assuming a 4/4 key
          "h":1/2, # half-measure
          "q":1/4, # quarter-measure}

Class Composer:
  #...
  def create_sequence(self, sequence):
    #...
    dur_seq, sym_seq = self.split_sequence(sequence)
    #...
  def split_sequence(self, sequence):
    k=0; sym_seq=[]; dur_seq=[]
    while k<len(sequence):
      sym=sequence[k]
      k+=1
      sym_seq.append(sym)
      dur_seq.append(sym)
      k+=1
    return dur_seq, sym_seq
```

GRAMMAR-BASED COMPOSITION

Ex 1, i.e., dummy exercise

- Download the drums.zip package from webeep and copy its content in the folder "sounds"
- Choose a sound (or copy a new one) and use it in the composition
- Choose a bpm
- Execute the script and listen to your first grammar-based composition

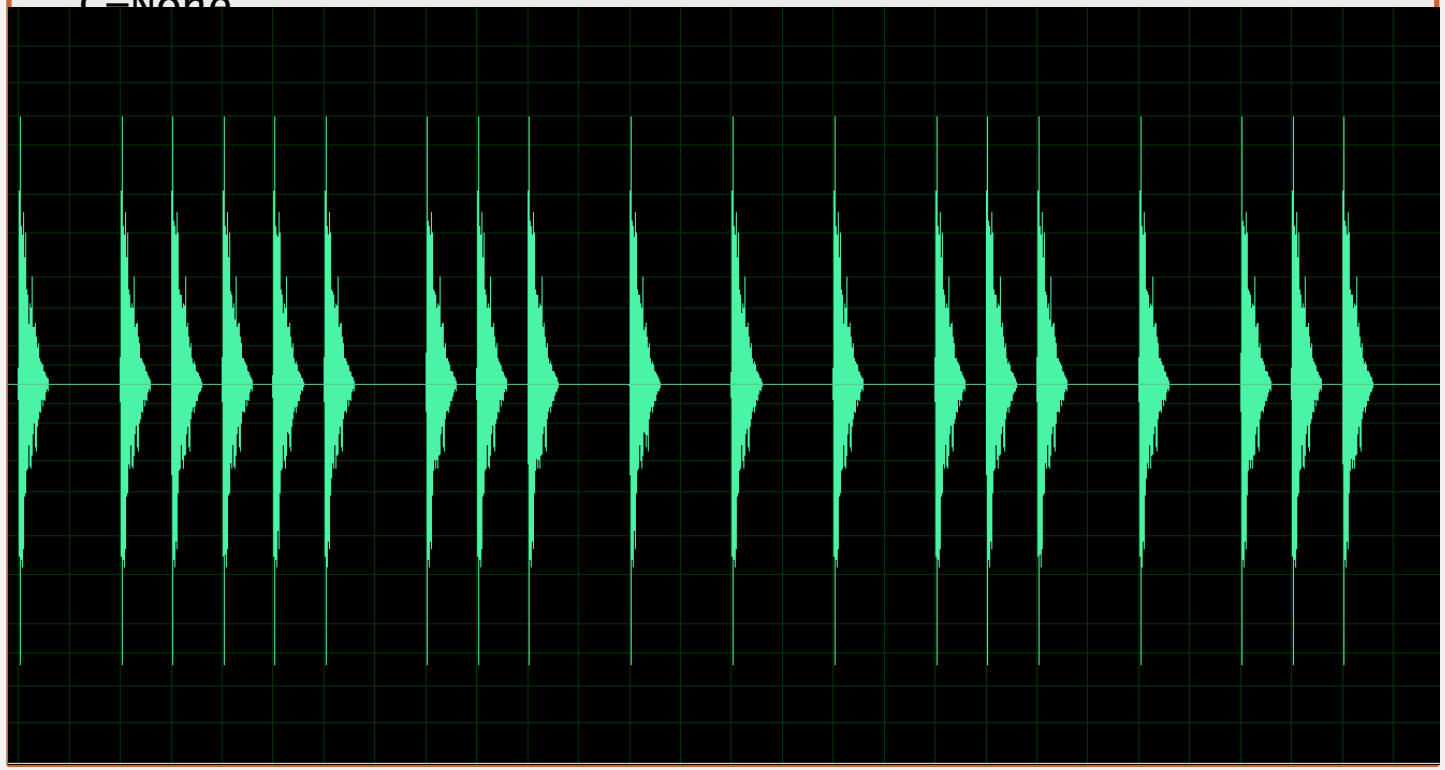
```
# ex1.py
if __name__=="__main__":
  EX=1
  C=None
  NUM_MEASURES=8
  START_SEQUENCE="M"*NUM_MEASURES# 8 measure
  MONO_COMPOSITION=EX<7 # first 6 exercises are mono
  if EX==1:
    G=Grammar_Sequence(basic_grammar)
    fn_out="basic_composition.wav"
  if MONO_COMPOSITION:
    seqs=G.create_sequence(START_SEQUENCE)
    print("\n".join(seqs))
    print("\nFinal sequence: ", G.sequence)
    bpm= <your bpm>
    C= Composer("sounds/<your sound>.wav", BMP=bpm)
    C.create_sequence(G.sequence)
    C.write("out/"+fn_out)
```

GRAMMAR-BASED COMPOSITION

Ex 1, i.e., dummy exercise

- Download the drums.zip package from webeep and copy its content in the folder "sounds"
- Choose a sound (or copy a new one) and use it in the composition
- Choose a bpm
- Execute the script and listen to your first grammar-based composition

```
# ex1.py
if __name__=="__main__":
  EX=1
  C=None
```



GRAMMAR-BASED COMPOSITION

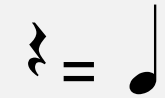
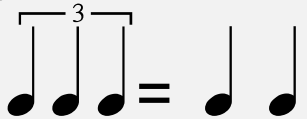
Ex 2, with octave notes

1. Include the word **o** in the `word_dur` variable
   - pass it to the `composer init`
2. Create the grammar `octave_grammar` supporting **o**, possibly adding new non-terminal symbols
   - what "oo" should be equal to?
3. Create the new composition, possibly changing the instrument

```
# ex2.py
octave_grammar={
    # your code here...
}

o_word_dur={"h":0.5, "q":0.25,
           # your code here: add "o"
  }

if __name__=="__main__":
  fn_out="octave_composition.wav"

  NUM_M=8
  START_SEQUENCE="M"*NUM_M
  G=Grammar_Sequence(octave_grammar)

  seqs=G.create_sequence(START_SEQUENCE)
  C= Composer("sounds/cymb.wav",
                  o_word_dur, BPM=174)
  C.create_sequence(G.sequence)
  C.write("out/"+fn_out)
```

GRAMMAR-BASED COMPOSITION

Ex 3, triplets and pauses

1. Include the words ***${h,q,o}*** and ***t{h,q,o}*** in `word_dur`, to support pauses and triplets, respectively
   - ***$q*** has the same duration of *q* 
   - ***tqtqtq*** has the same duration of *qq* 
2. Create the grammar `triplet_grammar` to supporting the new words
   - E.g. **H** → 
3. Create the new composition

```
# ex3.py
triplet_grammar={ # your code here }
triplet_word_dur={# your code here }

if __name__=="__main__":
  fn_out="triplet_composition.wav"
  G=Grammar_Sequence(triplet_grammar)
  C= Composer("sounds/cymb.wav",
                triplet_word_dur)
   # ...
  C.create_sequence(G.sequence)
  C.write("out/"+fn_out)
```

GRAMMAR-BASED COMPOSITION

Ex 4, slow down

If we want to use a bass drum, we don't want to play fast octave triplets, so let's create a slow grammar

1. Include the word ***w*** in the `word_dur` variable to support a whole 𝅝
   - Include the corresponding pause
2. Create the grammar `slow_grammar` supporting ***w***, and remove notes that are too short
3. Create the new composition, using an appropriate instrument

```
# ex4.py
slow_grammar={ #your code here }
slow_word_dur= {# your code here    }

if __name__=="__main__":
  fn_out="slow_composition.wav“
  # ...
  G=Grammar_Sequence(slow_grammar)
  C= Composer("sounds/kick.wav",
                slow_word_dur)
  # ...
```

GRAMMAR-BASED COMPOSITION

Ex 5, add syncopation

We are using regular times . Let's spice up things

1. Include the words ***p{h,q,o}*** in `word_dur`, to support pauses of triplets
   - ***pq*** has the same duration of t*q*
2. create the grammar `upbeat_grammar`
   - supporting ***p{h,q,o}***
   - removing structures that are too regular
   - Supporting upbeats and syncopation
- For example:

H → 

```
# ex5.py
upbeat_grammar={ #your code here }
upbeat_word_dur= {# your code here    }

if __name__=="__main__":
  fn_out="upbeat_composition.wav"
  # ...
  G=Grammar_Sequence(upbeat_grammar)
  C= Composer("sounds/upbeat.wav",
                upbeat_word_dur)
  # ...
```

GRAMMAR-BASED COMPOSITION

Short break: how to add new symbols?

Look at method `split_sequence` from class `Composer` in `classes.py`

- If a new character is $, t or p → there is a two-letter word
- else, it is a one-letter word

If you want to add yet another two-letter word with a new symbol, modify the string `"$tp"`.

E.g. dh= dotted-half= 

```
# classes.py
def split_sequence(self, sequence):
  k=0; sym_seq=[]; dur_seq=[]
  while k<len(sequence):
    if sequence[k] in "$tp":
      sym = sequence[k] + sequence[k+1]
      k+=2
    else:
      sym=sequence[k]
      k+=1
    sym_seq.append(sym)
    dur_seq.append(self.word_dur[sym])
  return dur_seq, sym_seq
```

GRAMMAR-BASED COMPOSITION

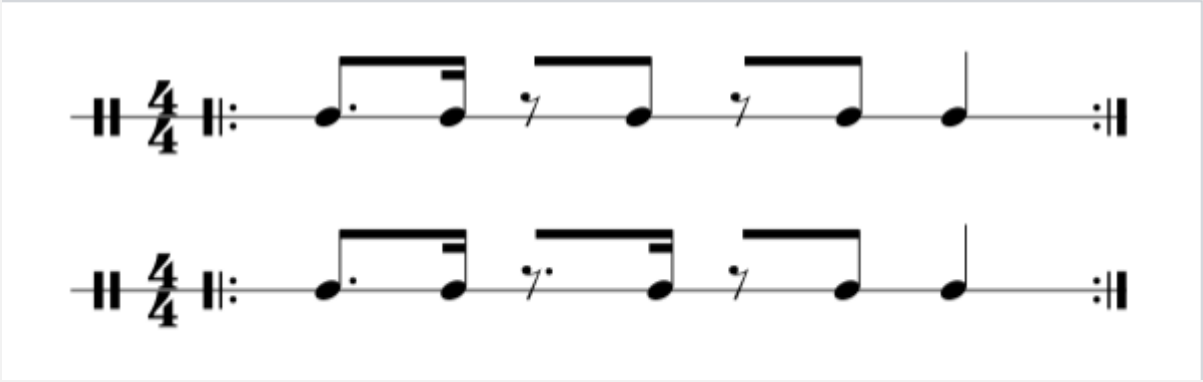
Ex 6, what about the clave?

```
# ex6.py
if __name__=="__main__":
  fn_out="clave_composition.wav"
  G=Grammar_Sequence(clave_grammar)
```

- *The clave is a rhythmic pattern used as a tool for temporal organization in Afro-Cuban music. It is present in a variety of genres such as Abakuá music, rumba, conga, son, mambo, salsa, songo, timba and Afro-Cuban jazz. The five-stroke clave pattern represents the structural core of many Afro-Cuban rhythms.*

https://en.wikipedia.org/wiki/Clave_(rhythm)

- Create a `clave_grammar` that randomly chooses between a son clave (top) and a rumba clave (bottom)



GRAMMAR-BASED COMPOSITION

```
# main.py
# variable names were shortened
class Track:
  def __init__(self, composer,
                     grammar, gain, sr):
    self.C=composer; self.sr=sr
    self.G=grammar; self.gain=gain
  def create_sequence(self, start_seq):
    self.G.create_sequence(start_seq)
    self.C.create_sequence(self.G.sequence)

def write_mix(tracks, fn_out="out.wav"):
  maxN=0
  for t in tracks:
    maxN=max(maxN, t.C.sequence.size)
  track=np.zeros((len(tracks), maxN))
  for i, t in enumerate(tracks):
    seq= t.C.sequence; N=seq.size
    track[i, :N]=t.gain*seq
  track = np.sum(track, axis=0)
  track = 0.707*track/np.max(np.abs(track))
  sf.write(fn_out, track, t.sr)
```

Ex 7, create a mix

1. Create your mix choosing a sample, a grammar and a gain for each *track*
2. See the `Track` class, characterized by: a Grammar, a Composer, a `gain`
   - Use `Track.create_sequence`
3. Use `write_mix` with a list of tracks

```
samples=[#your samples]
grammars=[# your grammars]
gains = [ #your gains]
tracks=[]

for i in range(len(samples)):
   # your code here
```

GRAMMAR-BASED COMPOSITION

Defining a grammar to perform a composition involves to define a vocabulary, a set of rules and a starting sentence.

We drew words as in a uniform distribution, but we can make some words be more probable than others.

You can compose a melody using grammars where words are pitches, and not only rhythmic figures.

PYTHON – CLIENT-SERVER

PYTHON – CLIENT-SERVER

We can use OSC to make different apps to communicate and interact with each other

What if you want to make other people to interact with your app?

- We can use a OSC-based app, but OSC does not scale well with multiple people
- We can use web apps, which allow users to just navigate to your browser
- We need a server to collect answers from users and tell them back to our app

Server-side programming can be used for several other purposes in your installation:

- Make apps communicate with each other even if they are not connected to the same network
- Distribute your installation to the world

PYTHON – CLIENT-SERVER

- Python can be used to call remote servers for applications
  - It can also be used to write server-app for remote computation
  - High-level language and server-side power
- In this course we will use PythonAnywhere.com
  - Because it's free
  - Other services include common cloud providers such as Google App Engine, Amazon Web Services, Microsoft Azure, Heroku, etc

PYTHON – CLIENT-SERVER

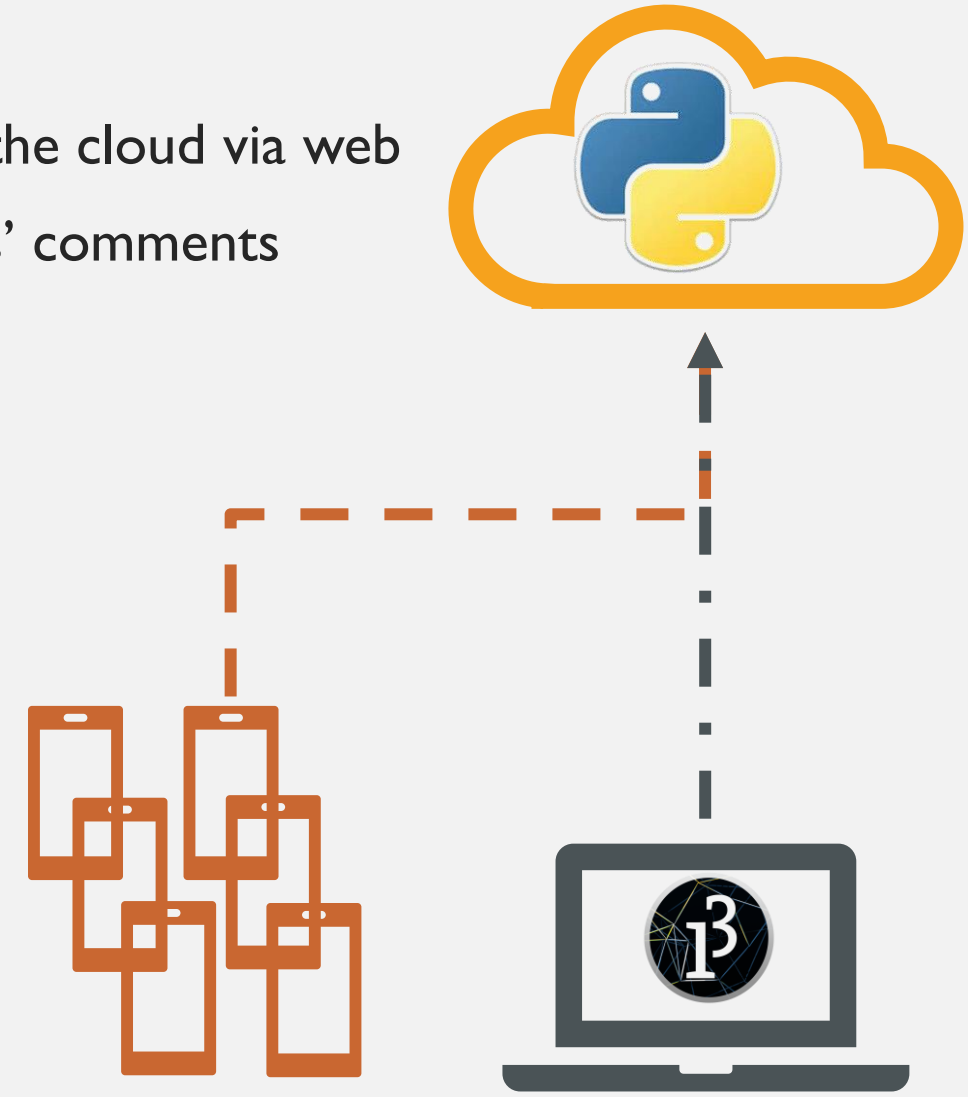
In our first app, we will replicate a visual performance that Zanoni/Buccoli developed in 2020 for the *Prophet in the Wind* flute performance

- At the end of a flute concert, people in the audience are asked to share their feeling through a web app, i.e., #shareyourheart
- On the stage the words shared by the audience are projected on a wall through a particle system
  - Here we will use a simpler visualiser

PYTHON – CLIENT-SERVER

The architecture needs to support

- Multiple clients (commonly smartphones) connecting to the cloud via web
- A "special" client running Processing to receive audiences' comments

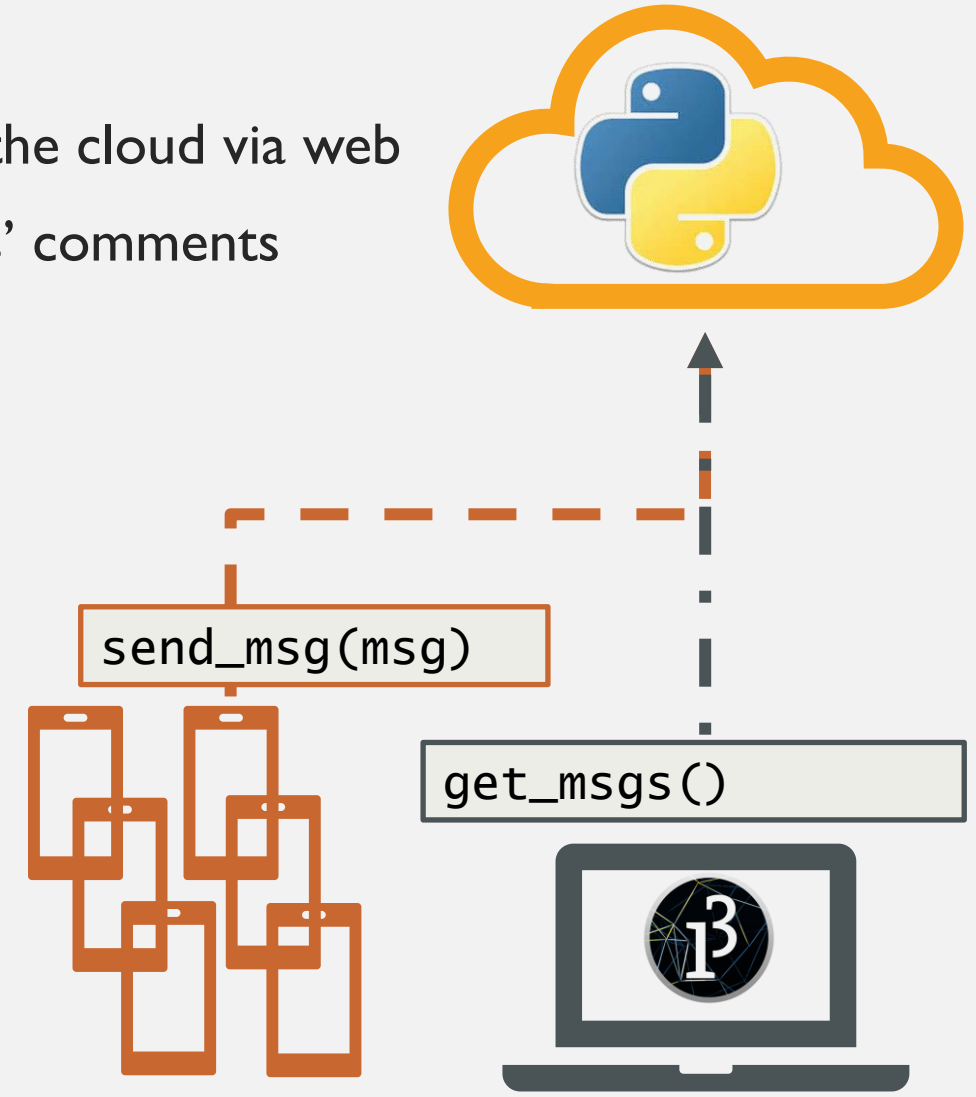


PYTHON – CLIENT-SERVER

The architecture needs to support

- Multiple clients (commonly smartphones) connecting to the cloud via web
- A "special" client running Processing to receive audiences' comments

- We need to deploy two APIS:
  - `send_msg(msg)` send the message
  - `get_msgs()` get all the messages

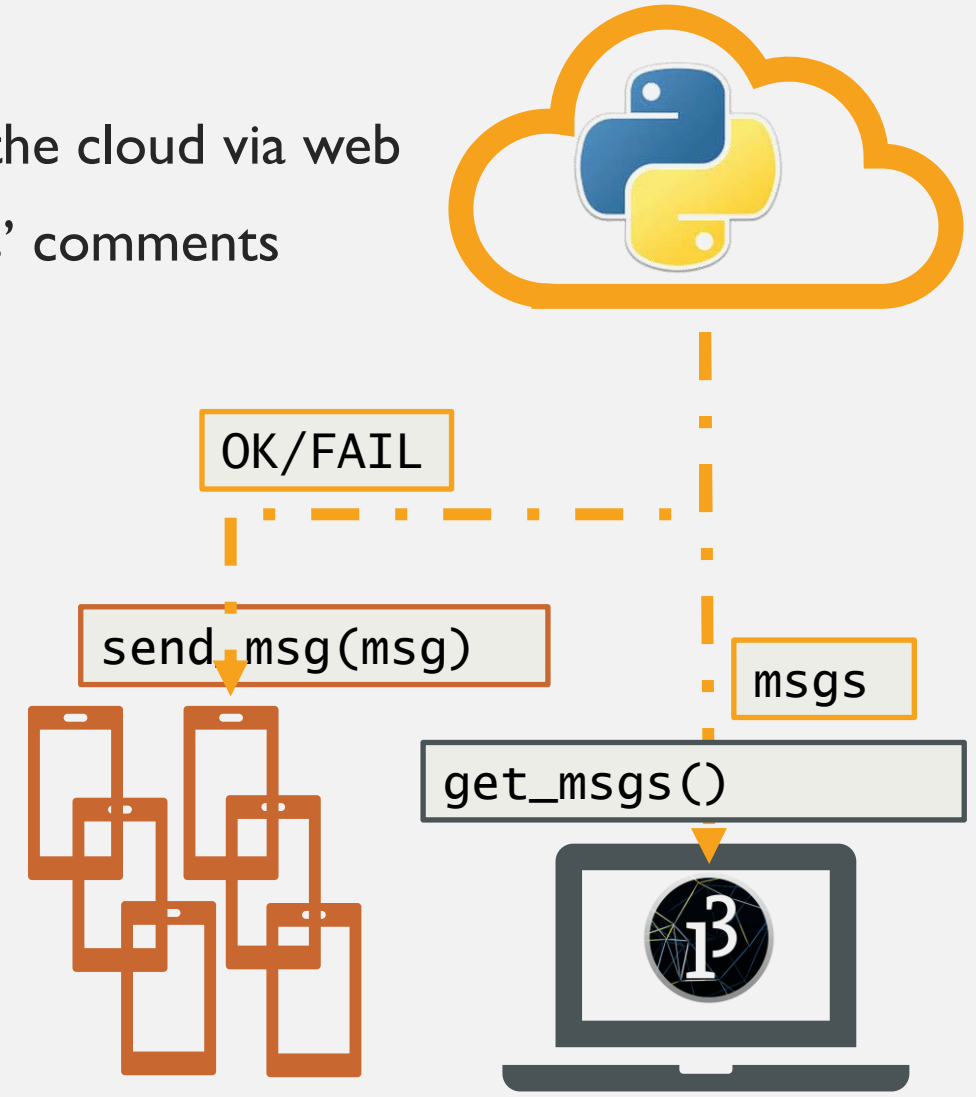


PYTHON – CLIENT-SERVER

The architecture needs to support

- Multiple clients (commonly smartphones) connecting to the cloud via web
- A "special" client running Processing to receive audiences' comments

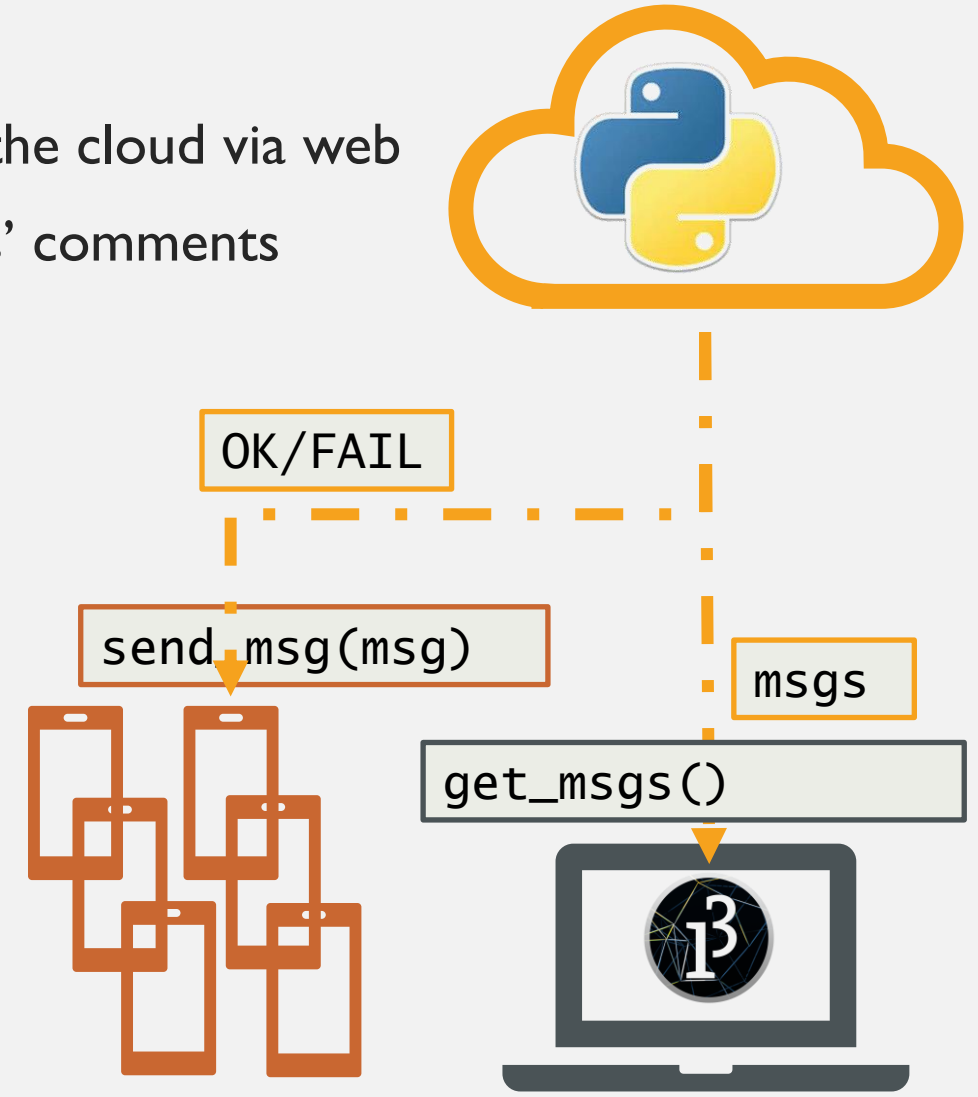
- We need to deploy two APIS:
  - `send_msg(msg)` send the message
    - The server returns a OK/FAIL status
  - `get_msgs()` get all the messages
    - The server returns the list of messages



PYTHON – CLIENT-SERVER

The architecture needs to support

- Multiple clients (commonly smartphones) connecting to the cloud via web
- A "special" client running Processing to receive audiences' comments



- We need to deploy two APIS:
  - `send_msg(msg)` send the message
    - The server returns a OK/FAIL status
  - `get_msgs()` get all the messages
    - The server returns the list of messages

- At the end of these slides, there is a tutorial on how to create the test using Django framework

PYTHON – CLIENT-SERVER

- PythonAnywere allows to directly edit the files
  - This is actually *not* the right way to code on server
- The **right way** involves to
  - Start a repository
  - Implement the server *locally* in your computer
  - Do an extensive set of tests to catch possible errors
  - Deploy your code remotely by using appropriate tools or by cloning the repository in the server

- For the sake of brevity, the tutorial will show how to edit files online

PYTHON – CLIENT-SERVER

After the tutorial, we have two APIs (use <zabucco> instead of <user> for our exercises)

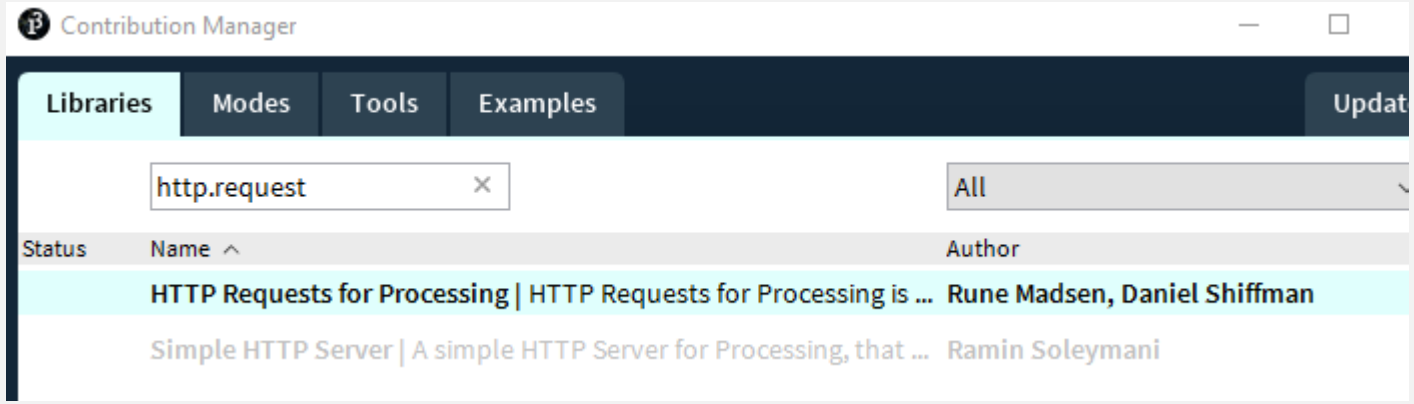
1. http://<user>.pythonanywhere.com/set_msg?text=inspired
   Send a message as a querystring (in the example above, the message is *inspired*)
   - Answer: `{"status": "ok", "message": "ok"}`
2. http://<user>.pythonanywhere.com/get_msgs
   Ask all the current messages, sorted by date
   - Answer: `{"msgs": ["inspired"]}`

PYTHON – CLIENT-SERVER

Now we need to create the Processing client!

- Every N seconds it needs to check the presence of the items
- Then it parses and draws what appears
- Dummy code, by now

- We will need the HTTP requests library for Processing
  - Sketch → Import library → add library



PYTHON – CLIENT-SERVER

```
# api_client.pde
import http.requests.*;
class API_Client{
  GetRequest req;
  String get_msg_api="";
  API_Client(String mainUrl){
    this.get_msg_api=mainUrl+"/get_msgs";
    this.req = new GetRequest(this.get_msg_api);
  }
  String[] get_msgs(){
    this.req.send();
    JSONObject JSONobj = parseJSONObject(req.getContent());
    JSONArray JSONmsgs = JSONobj.getJSONArray("msgs");
    String[] msgs=new String[JSONmsgs.size()];
    for (int t=0; t<JSONmsgs.size(); t++){
      msgs[t] = JSONmsgs.getString(t);
    }
    return msgs;
  }
}
```

We use a class API_Client to collect the data

PYTHON – CLIENT-SERVER

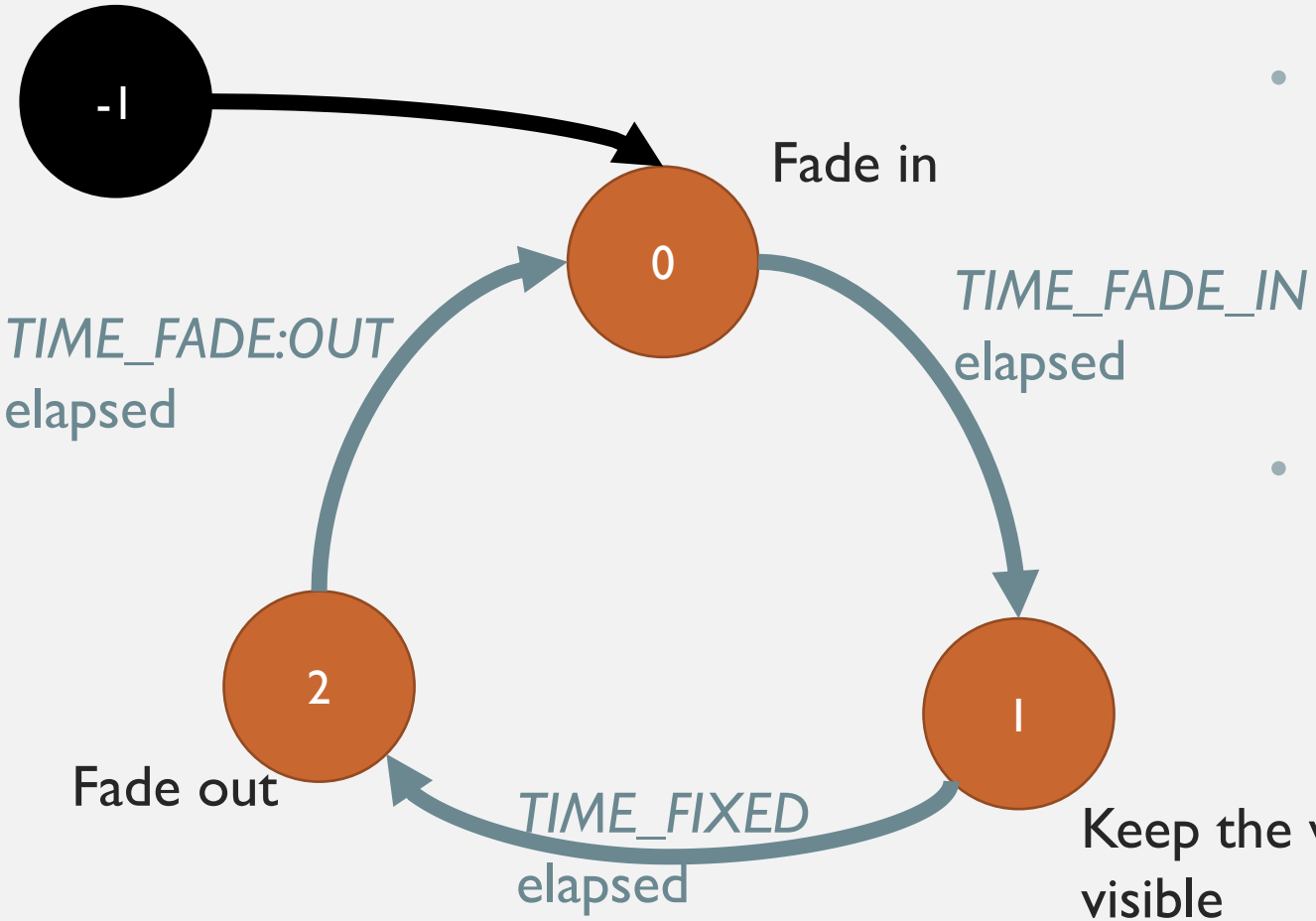
```
# api_client.pde
import http.requests.*;
class API_Client{
  GetRequest req;
  String get_msg_api="";
  API_Client(String mainUrl){
    this.get_msg_api=mainUrl+"/get_msgs";
    this.req = new GetRequest(this.get_msg_api);
  }
  String[] get_msgs(){
    this.req.send();
    JSONObject JSONobj = parseJSONObject(req.getContent());
    JSONArray JSONmsgs = JSONobj.getJSONArray("msgs");
    String[] msgs=new String[JSONmsgs.size()];
    for (int t=0; t<JSONmsgs.size(); t++){
      msgs[t] = JSONmsgs.getString(t);
    }
    return msgs;
  }
}
```

We use a class API_Client to collect the data

Convert from json object to Array of strings

PYTHON – CLIENT-SERVER

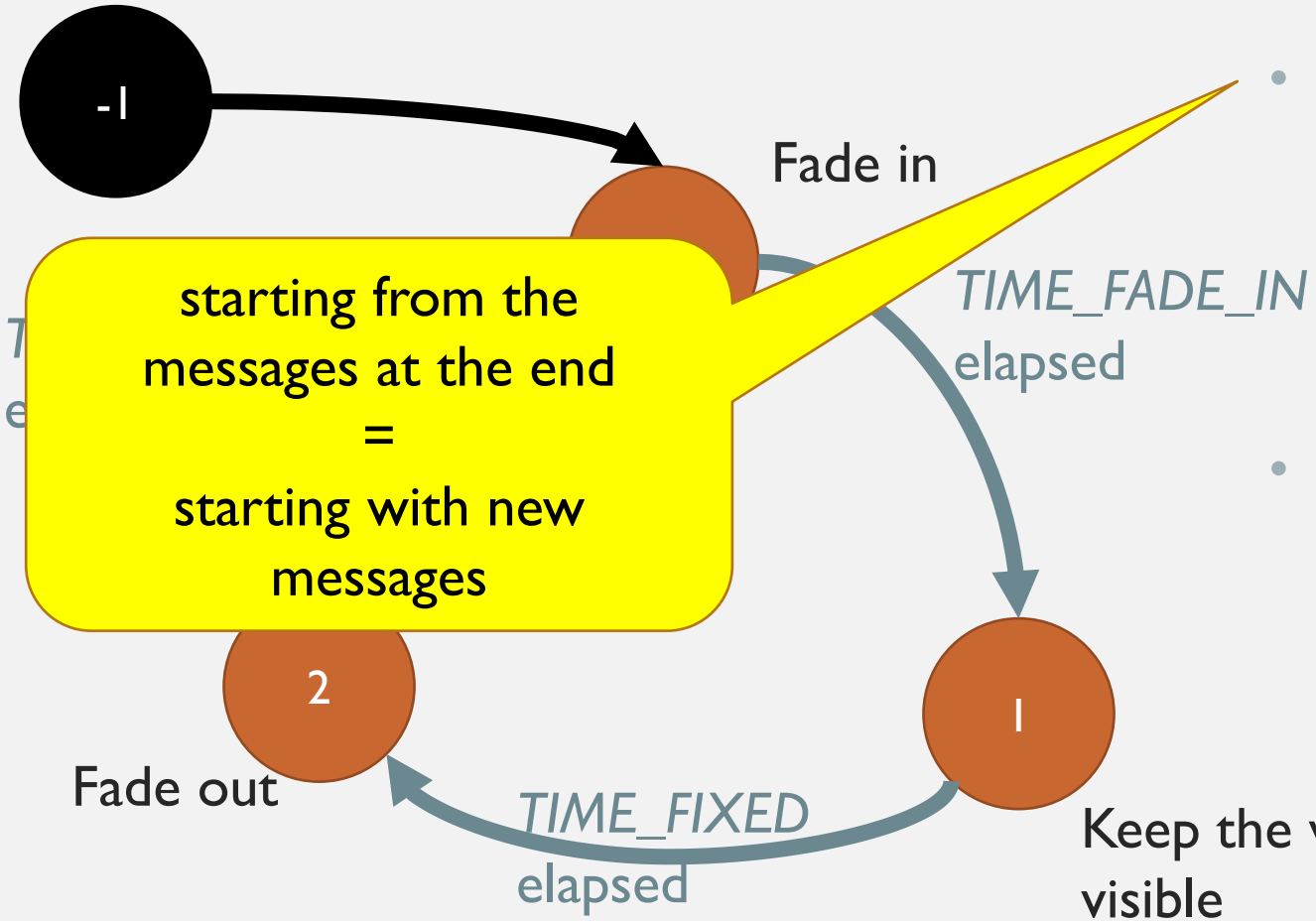
We use a state machine to pass between words



- Get the messages and take the last message
  - Make it fade in, stay, fade_out
  - If there are other messages, take the previous one
- Else, get new messages
  - If there are no messages, wait before asking again

PYTHON – CLIENT-SERVER

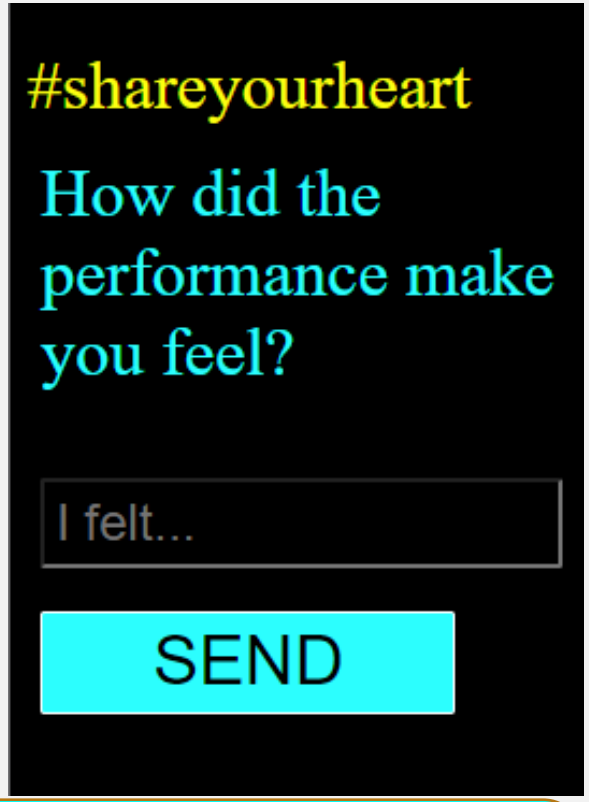
We use a state machine to pass between words



- Get the messages and take the last message
  - Make it fade in, stay, fade_out
  - If there are other messages, take the previous one
- Else, get new messages
  - If there are no messages, wait before asking again

PYTHON – CLIENT-SERVER

- If you need to use a python server-based implementation, you will also have to create a front-end for users to connect to.
- There are several ways to do, more or less secure
- In Javascript, using JQuery, you can write like in the box
  - However, it is safer to use nodejs to not expose the APIs
  - Or assign every page a token so only web-pages with such tokens can contact your API



Submit is the function assigned to a button click in the HTML

Get the text of the message from a form with id #textForm

```
# client.js
function submit(){
  let data={"text":$("#textForm")[0].value};
  $.getJSON(URL+"send_msg", data=data, showRefresh);
}
```

Make the request asking for a JSON; it will redirect the answer to the function showRefresh

PYTHON – CLIENT-SERVER

- Exercize 1: Let's use it together
  - Teacher shows the processing
  - You use the api to answer the question: "how did this course make you feel?"
    http://<user>.pythonanywhere.com/set_msg?text=tgif

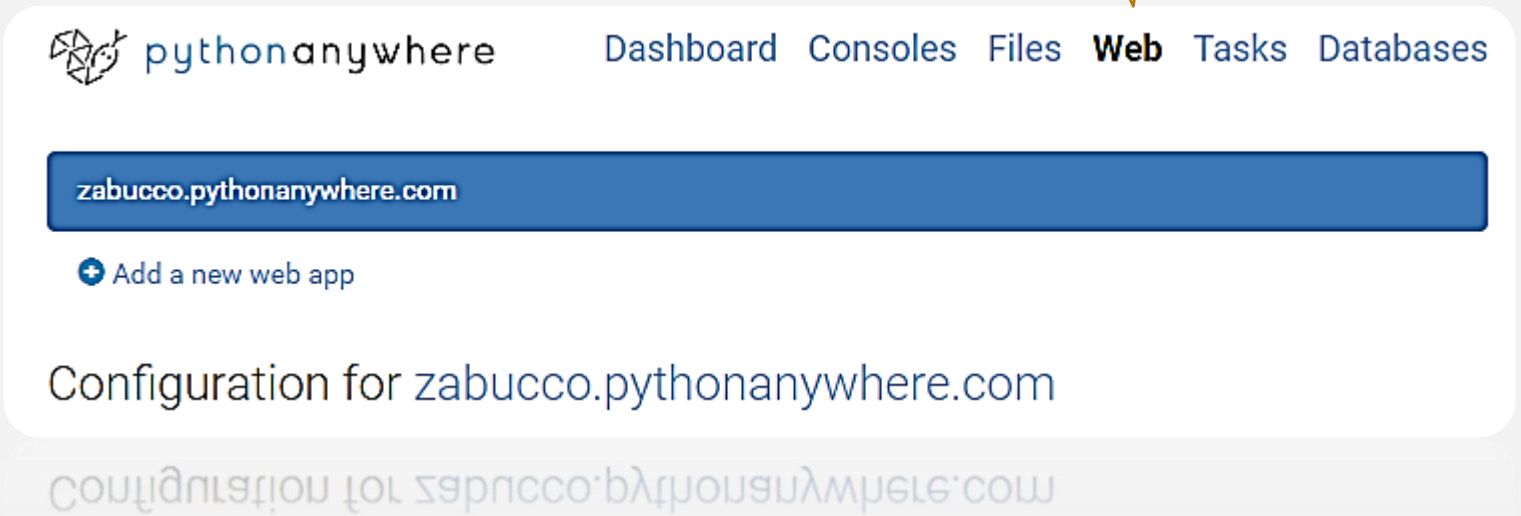
- Exercize 2: make the Processing visualizer more interesting, such as:
  - Change the color of Text randomly or dynamically
  - Show the text on top of some other processing script of the previous lessons
    - Text on top of a Levy flight, maybe?
  - Make some sound react to text
    - Each letter of the alphabet is a midinote between 60 and 86 ?

HOW TO CREATE A SERVER

PYTHON – CLIENT-SERVER

- Go to pythonanywhere.com and sign up
  - Use a good username, because it will become your domain name (free app)
- Open **WEB** tab → Add a new web app
  - Select a Python Web framework: Django!
  - Select a Python version: Python 3.7
  - Project name: shareyourheart
- Great! We can start

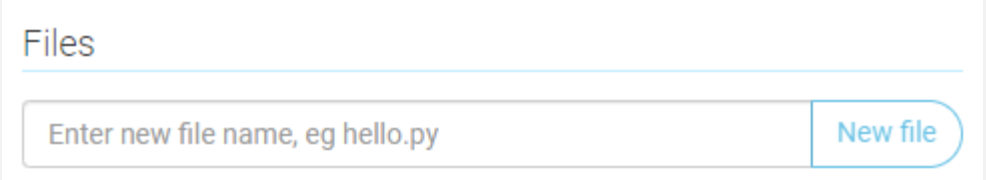
I will refer to these tabs with capital letters: **DASHBOARD, FILES, WEB**, etc…



PYTHON – CLIENT-SERVER

First thing first: the hello world!

- Go to the **FILES**/shareyourheart/shareyourheart
- New file: **views.py**; it contains the visualization
  - the index view is just a "Hello World" plain HTML
- Edit file **urls.py**
  - Add to the import to look into views




```
# views.py
from django.http import HttpResponse

def index(request):
  return HttpResponse("Hello World")
```

```
# urls.py
from django.contrib import admin
from django.urls import path
from shareyourheart import views

urlpatterns = [
  path('admin/', admin.site.urls),
  path('', views.index, name="index"),]
```

PYTHON – CLIENT-SERVER

First thing first: the hello world!

- Go to the **FILES**/shareyourheart/shareyourheart
- New file: **views.py**; it contains the visualization
  - the index view is just a "Hello World" plain HTML
- Edit file **urls.py**
  - Add to the import to look into views


```
# views.py
from django.http import HttpResponse

def index(request):
  return HttpResponse("Hello World")
```

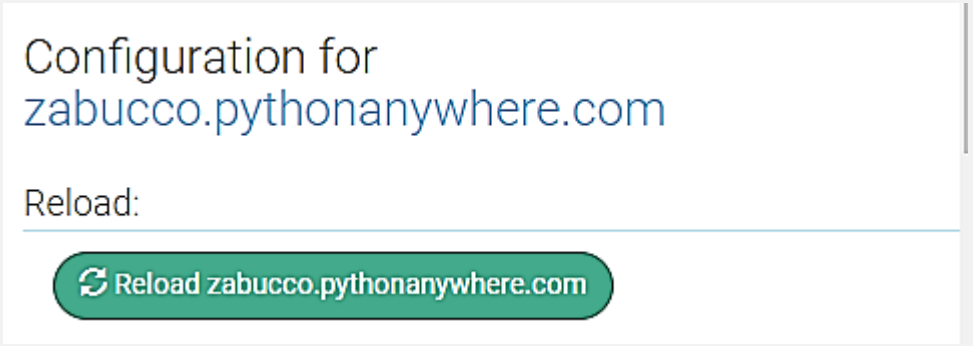
A request of URL as
name.pythonanywhere.com/<nothing>
(empty string)
must be handled by the function
index in views

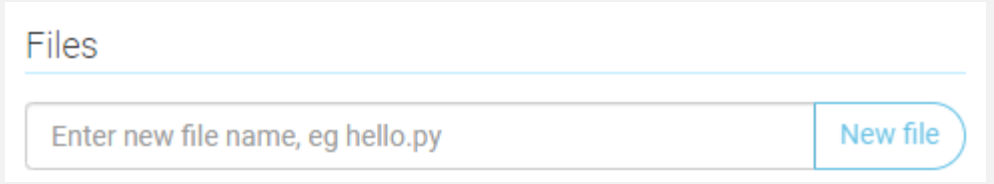
```
# urls.py
from django.contrib import admin
from django.urls import path
from shareyourheart import views

urlpatterns = [
  path('admin/', admin.site.urls),
  path('', views.index, name="index"),]
```

PYTHON – CLIENT-SERVER

First thing first: the hello world!

- Go to the **FILES**/shareyourheart/shareyourheart
- New file: **views.py**; it contains the visualization
  - the index view is just a "Hello World" plain HTML
- Edit file **urls.py**
  - Add to the import to look into views
- Back to **WEB,** *reload* and visit your website again





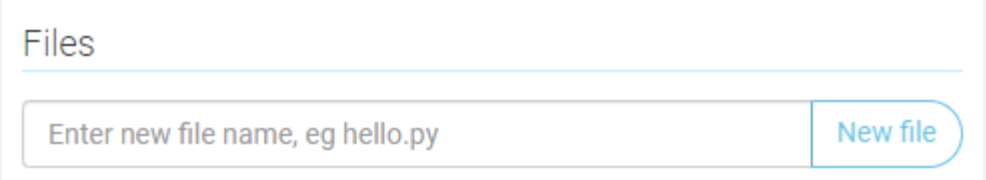

```python
# views.py
from django.http import HttpResponse

def index(request):
  return HttpResponse("Hello World")
```

```python
# urls.py
from django.contrib import admin
from django.urls import path
from shareyourheart import views

urlpatterns = [
  path('admin/', admin.site.urls),
  path('', views.index, name="index"),]
```

PYTHON – CLIENT-SERVER

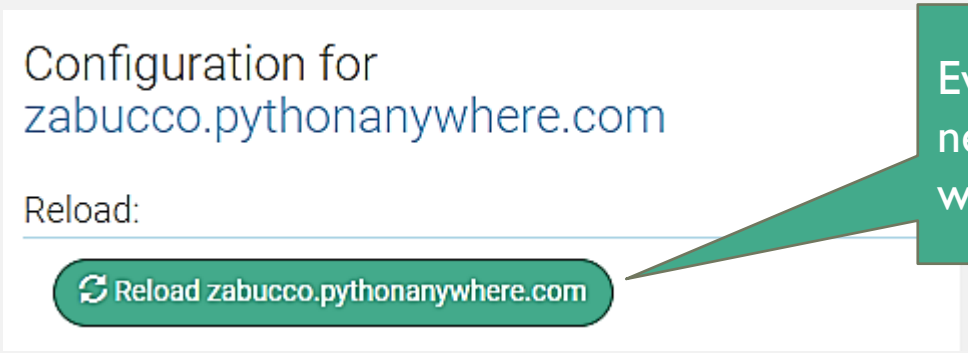
First thing first: the hello world!

- Go to the **FILES**/shareyourheart/shareyourheart
- New file: **views.py**; it contains the visualization
  - the index view is just a "Hello World" plain HTML
- Edit file **urls.py**
  - Add to the import to look into views
- Back to **WEB,** *reload* and visit your website again




```
# views.py
from django.http import HttpResponse

def index(request):
  return HttpResponse("Hello World")
```

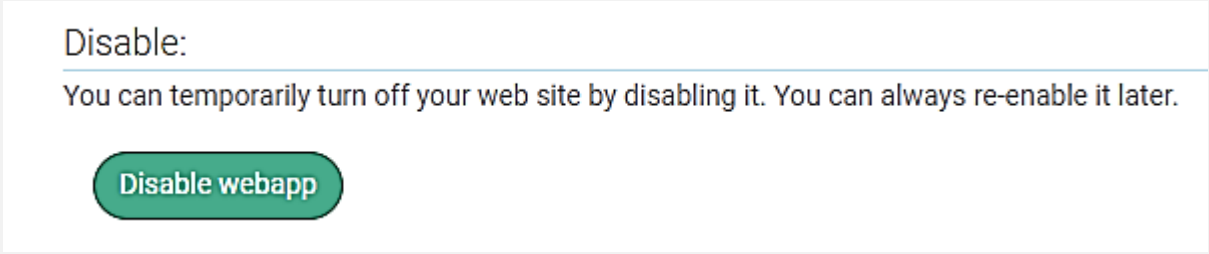
```
# urls.py
from django.contrib import admin
from django.urls import path
from shareyourheart import views

                                min.site.urls),
                                dex, name="index"),]
```



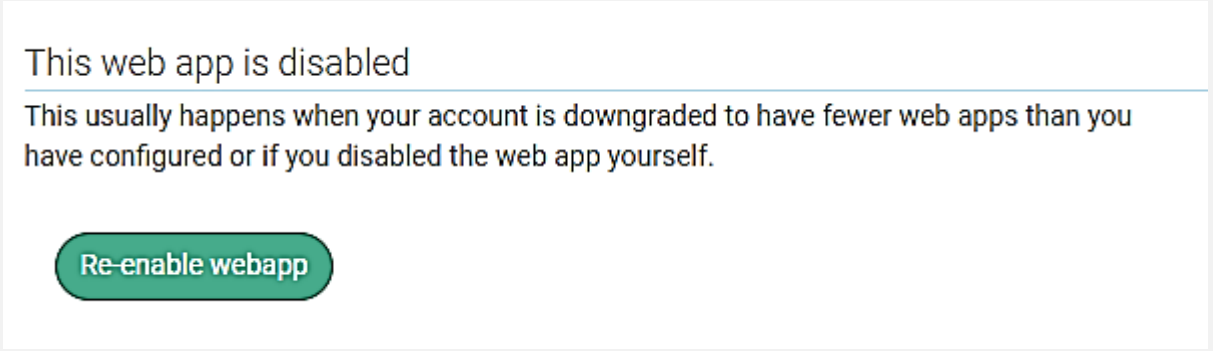
Every time you do some changes, you need to tell your server to reload the webapp so it can deploy them

PYTHON – CLIENT-SERVER

- You can always temporarily disable your web app

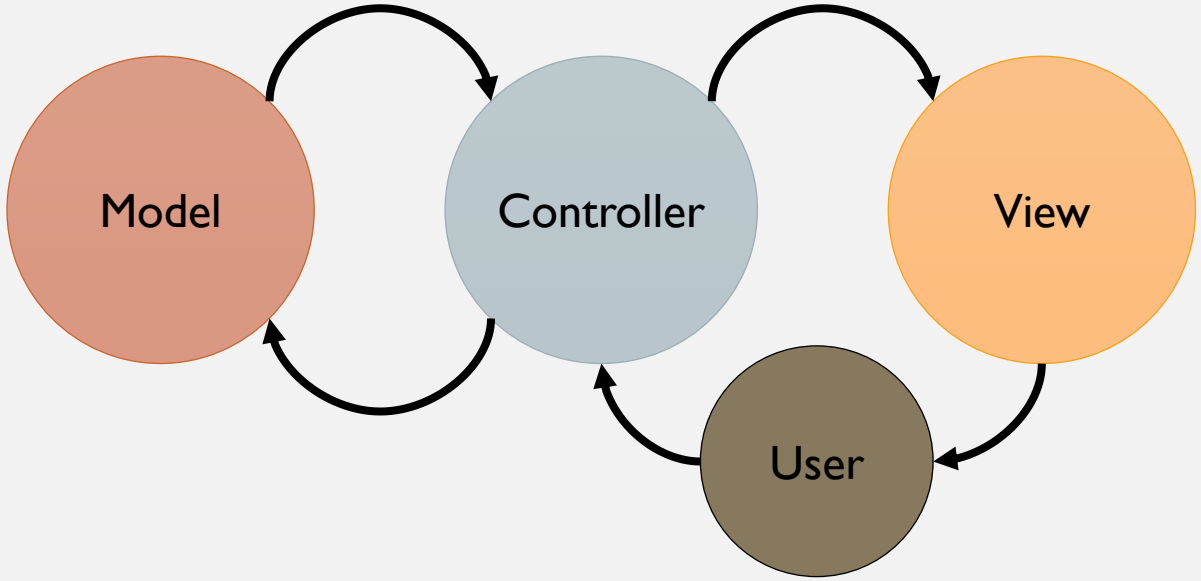


- So people cannot use it when they are not supposed to



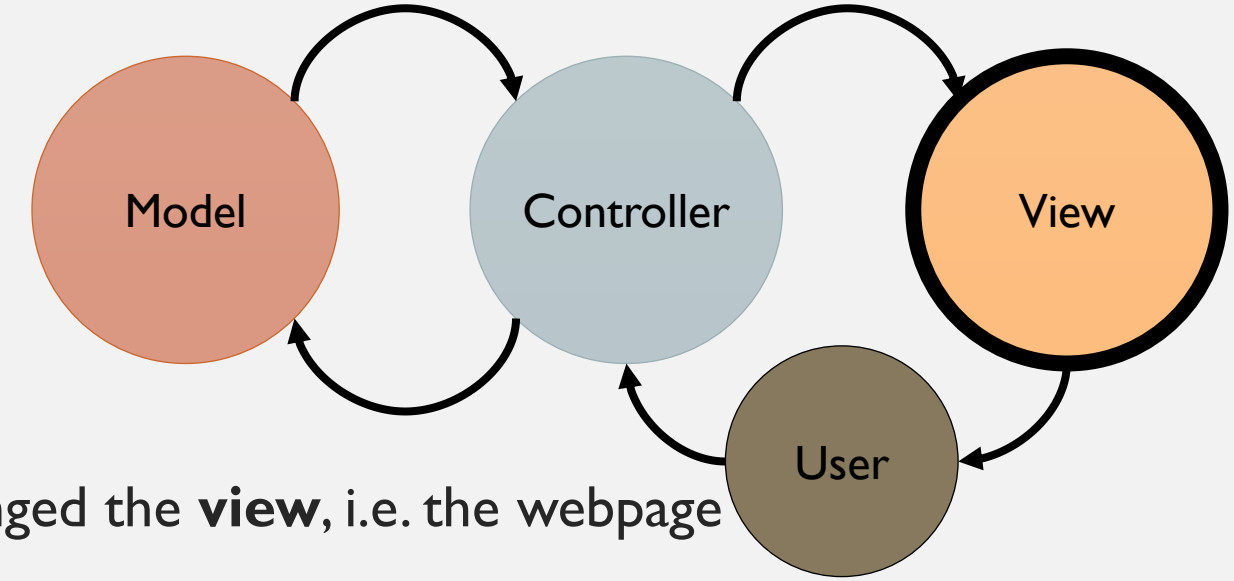
PYTHON – CLIENT-SERVER

- The Django framework relies on the **model-view-controller** design pattern
  - Model: where we store data
  - View, where we represent the world
  - Controller: interacts with user and changes apply changes
  - The Model and View do not communicate directly



PYTHON – CLIENT-SERVER

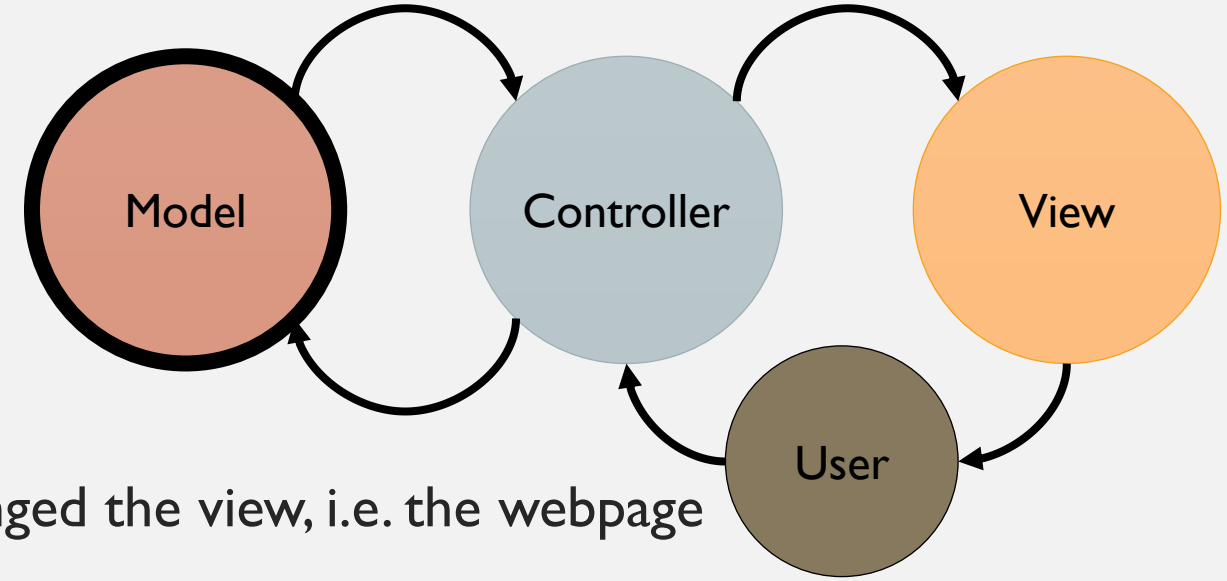
- The Django framework relies on the **model-view-controller** design pattern
  - Model: where we store data
  - View, where we represent the world
  - Controller: interacts with user and changes apply changes
  - The Model and View do not communicate directly



- In the previous examples, we changed the **view**, i.e. the webpage

PYTHON – CLIENT-SERVER

- The Django framework relies on the **model-view-controller** design pattern
  - Model: where we store data
  - View, where we represent the world
  - Controller: interacts with user and changes apply changes
  - The Model and View do not communicate directly



- In the previous examples, we changed the view, i.e. the webpage
- Now we need to build the **model**, i.e. the data we need to store
  - i.e., the string they include

PYTHON – CLIENT-SERVER

- **FILES**/shareyourheart/shareyourheart
- New file: **models.py**→

```
# models.py
from django.db import models as M
MAX_LENGTH=50

class Text(M.Model):
  text = M.CharField(max_length=MAX_LENGTH)
```

«CharField» is a string
https://docs.djangoproject.com/en/3.1/topics/db/models/

PYTHON – CLIENT-SERVER

- **FILES**/shareyourheart/shareyourheart
- New file: **models.py**→
- Edit **settings.py** and add shareyourheart in the INSTALLED_APPS

```python
# models.py
from django.db import models as M
```

```python
# settings.py
"""
Django settings for shareyourheart project.
...
"""
# Application definition
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'shareyourheart',
]
```

PYTHON – CLIENT-SERVER

- **FILES**/shareyourheart/shareyourheart
- New file: **models.py**→
- Edit **settings.py** and add shareyourheart in the INSTALLED_APPS
- Go to CONSOLE/bash and type

```
# models.py
from django.db import models as M
```

```
# settings.py
"""
Django settings for shareyourheart project.
...
"""
# Application definition
```

```
$ cd shareyourheart
$ python3 manage.py makemigrations shareyourheart
$ python3 manage.py migrate
$ python3 manage.py shell
In [1]: from shareyourheart.models import Text
In [2]: Text.objects.all()
Out[2]: <QuerySet []>
```

PYTHON – CLIENT-SERVER

- **FILES**/shareyourheart/shareyourheart
- New file: **models.py**→
- Edit **settings.py** and add shareyourheart in the INSTALLED_APPS
- Go to CONSOLE/bash and type

You need to run the makemigrations command everytime you make some changes in the model structure

```
# models.py
from django.db import models as M
```

```
# settings.py
"""
Django settings for shareyourheart project.
...
"""

# Application definition
```

```
$ cd shareyourheart
$ python3 manage.py makemigrations shareyourheart
$ python3 manage.py migrate
```

```
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'shareyourheart',
]
```

PYTHON – CLIENT-SERVER

- **FILES**/shareyourheart/shareyourheart
- New file: **models.py**→
- Edit **settings.py** and add shareyourheart in the INSTALLED_APPS
- Go to **CONSOLES**/bash and type

```
# models.py
from django.db import models as M
```

```
# settings.py
"""
Django settings for shareyourheart project.
...
"""
# Application definition
```

You need to run the makemigrations command everytime you make some changes in the model structure

The shell allows you to see the Text model has been created and it is currently empty

```
$ cd shareyourheart
$ python3 manage.py makemigrations shareyourheart
$ python3 manage.py migrate
$ python3 manage.py shell
In [1]: from shareyourheart.models import Text
In [2]: Text.objects.all()
Out[2]: <QuerySet []>
```

PYTHON – CLIENT-SERVER

- Continue in the Console

```
In [1]: from shareyourheart.models import Text
In [2]: Text.objects.all()
Out[2]: <QuerySet []>
In [3]: t = Text()
In [4]: t.msg="well"
In [5]: t.save()
In [6]: Text.objects.all()
Out[6]: <QuerySet [<Text object>]>
In [7]: t.delete()
In [8]: Text.objects.all()
Out[8]: <QuerySet []>
In [9]: exit()
```

Create a new text, assigning an id and saving it

Now the database contains one object

Removing the object → the database is empty

- Go to **CONSOLES** and kill the bash (use x) to save computational time

PYTHON – CLIENT-SERVER

Now we can write the controller

We need two function:

1. send_msg(msg); save a new text; returns 200 if fine
2. get_msgs(); returns all the messages

We use JSON for output response

- **FILES**/shareyourheart/shareyourheart
- New file: **apps.py**→

```
# apps.py
from urllib.parse import parse_qs
from shareyourheart.models import Text
from django.http import JsonResponse

def get_msgs(request):
  texts=Text.objects.all()
  response={"msgs":[]}
  for text in texts:
    response["msgs"].append(text.msg)
  return JsonResponse(response)

def send_msg(request):
  try:
    query = parse_qs(request.META["QUERY_STRING"])
    obj=Text.objects.create(text=query["text"][0])
    obj.save()
    response={"status":"ok", "message":"ok"}
  except Exception as exc:
    response={"status":"fail", "message":str(exc)}
  return JsonResponse(response)
```

PYTHON – CLIENT-SERVER

Now we can write the controller

We need two function:

1. send_ms
   returns
2. get_ms

We use JSON for output response

- **FILES**/sh
- New file:

Just building the JSON response and returing it

Try/except: if something goes wrong, we can read what happened

We pass the text via query string

```
# apps.py
from urllib.parse import parse_qs
from shareyourheart.models import Text
from django.http import JsonResponse

def get_msgs(request):
  texts=Text.objects.all()
  response={"msgs":[]}
  for text in texts:
    response["msgs"].append(text.text)
  return JsonResponse(response)

def send_msg(request):
  try:
    query = parse_qs(request.META["QUERY_STRING"])
    obj=Text.objects.create(text=query["text"][0])
    obj.save()
    response={"status":"ok", "message":"ok"}
  except Exception as exc:
    response={"status":"fail", "message":str(exc)}
  return JsonResponse(response)
```

PYTHON – CLIENT-SERVER

Last steps:

- add the two APIs to urls.py →
- Reload the website (from **WEB**)

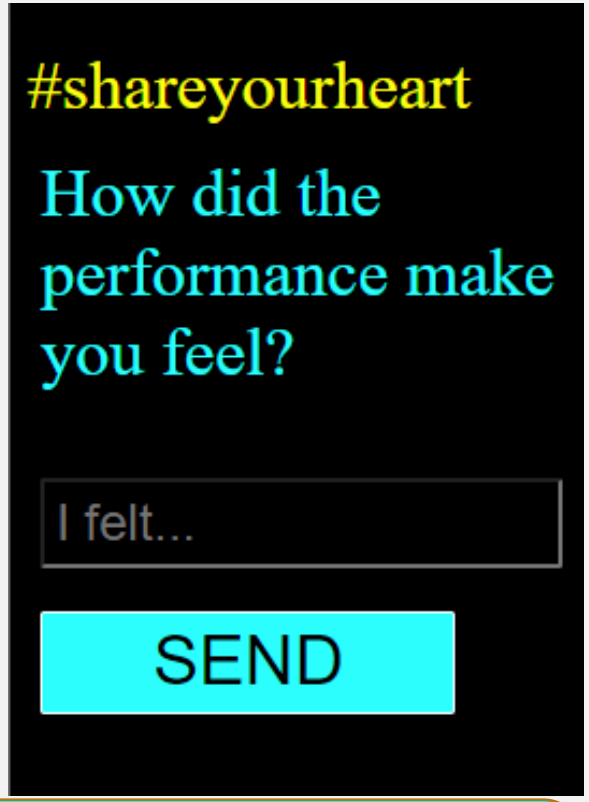

```
# urls.py
from django.contrib import admin
from django.urls import path
from shareyourheart import views
from shareyourheart import apps
urlpatterns = [
  path('admin/', admin.site.urls),
  path('', views.index, name="index"),
  path('send_msg', apps.send_msg, name="send_msg"),
  path('get_msgs', apps.get_msgs, name="get_msgs"),]
```

Let's test visiting these pages:

- http://<user>.pythonanywhere.com/get_msgs
  - Answer: `{"msgs": []}`
- http://<user>.pythonanywhere.com/set_msg?text=inspired
  - Answer: `{"status": "ok", "message": "ok"}`
- http://<user>.pythonanywhere.com/get_msgs
  - Answer: `{"msgs": ["inspired"]}`

PYTHON – CLIENT-SERVER

- If you need to use a python server-based implementation, you will also have to create a front-end for users to connect to.
- There are several ways to do, more or less secure
- In Javascript, using JQuery, you can write like in the box
  - However, it is safer to use nodejs to not expose the APIs
  - Or assign every page a token so only web-pages with such tokens can contact your API



Submit is the function assigned to a button click in the HTML

Get the text of the message from a form with id #textForm

```
# client.js
function submit(){
  let data={"text":$("#textForm")[0].value};
  $.getJSON(URL+"send_msg", data=data, showRefresh);
}
```

Make the request asking for a JSON; it will redirect the answer to the function showRefresh