[301] Dictionary Nesting

Tyler Caraza-Harter

Learning Objectives Today

More dictionary operations

- len, in, for loop
- d.keys(), d.values()
- defaults for get and pop

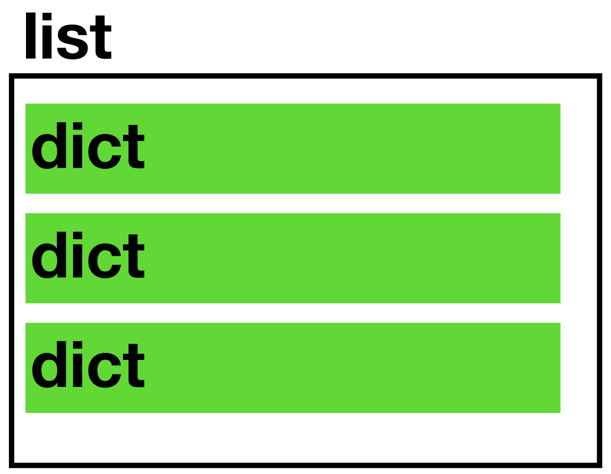
Learning Objectives Today

More dictionary operations

- len, in, for loop
- d.keys(), d.values()
- defaults for get and pop

Syntax for nesting (dicts inside dicts, etc)

- indexing/lookup
- step-by-step resolution



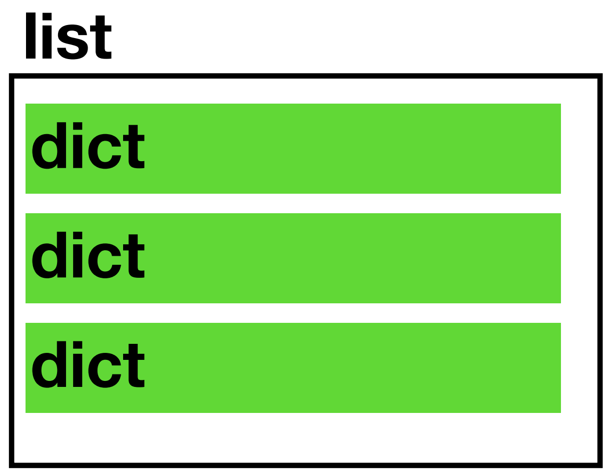
Learning Objectives Today

More dictionary operations

- len, in, for loop
- d.keys(), d.values()
- defaults for get and pop

Syntax for nesting (dicts inside dicts, etc)

- indexing/lookup
- step-by-step resolution



Understand common use cases for nesting

- binning/bucketing (list in dict)
- a more convenient table representation (dict in list)
- transition probabilities with Markov chains (dict in dict)

one of the most common data analysis tasks

we'll generate random English-like texts

Today's Outline

Dictionary Ops

Binning (dict of list)

Table Representation (list of dict)

Markov Chains (dict of dict)

Creation of Empty Dict

Non-empty dict:

```
d = {“a”: “alpha”, “b”: “beta”}
```

Empty dict (way 1):

```
d = {}
```

Empty dict (way 2):

```
d = dict()
```

Creation of Empty Dict

Non-empty dict:

```
d = {“a”: “alpha”, “b”: “beta”}
```

Empty dict (way 1):

```
d = {}
```

Empty dict (way 2):

```
d = dict()
```

similar for lists: `L = list()`

len, in, for

```
num_words = {0:“zero”, 1:”one”, 2:”two”, 3:”three”}


print(len(num_words))


print(1 in num_words)


print(“one” in num_words)


for x in num_words:
    print(x)
```

len, in, for

```
num_words = {0:“zero”, 1:”one”, 2:”two”, 3:”three”}

print(len(num_words))
```

$\Rightarrow$ **?**

```
print(1 in num_words)

print(“one” in num_words)

for x in num_words:
    print(x)
```

len, in, for

```
num_words = {0:“zero”, 1:”one”, 2:”two”, 3:”three”}

print(len(num_words))
```

4

```
print(1 in num_words)


print(“one” in num_words)


for x in num_words:
    print(x)
```

len, in, for

```
num_words = {0:“zero”, 1:”one”, 2:”two”, 3:”three”}

print(len(num_words))          →   4

print(1 in num_words)          →   ?

print(“one” in num_words)      →   ?

for x in num_words:
    print(x)
```

len, in, for

```
num_words = {0:“zero”, 1:”one”, 2:”two”, 3:”three”}

print(len(num_words))
```

4

```
print(1 in num_words)
```

True

```
print(“one” in num_words)
```

False

(it is only checking keys, not vals)

```
for x in num_words:
    print(x)
```

len, in, for

```
num_words = {0:“zero”, 1:”one”, 2:”two”, 3:”three”}
```

```
print(len(num_words))
```

4

```
print(1 in num_words)
```

True

```
print(“one” in num_words)
```

False

(it is only checking keys, not vals)

```
for x in num_words:
    print(x)
```

?
?
?
?

len, in, for

```
num_words = {0:“zero”, 1:”one”, 2:”two”, 3:”three”}
```

```
print(len(num_words))
```

4

```
print(1 in num_words)
```

True

```
print(“one” in num_words)
```

False

(it is only checking keys, not vals)

```
for x in num_words:
    print(x)
```

2
1
0
3

(for iterates over keys, not vals)

(note there is no order here)

len, in, for

```
num_words = {0:"zero", 1:"one", 2:"two", 3:"three"}
```

```
print(len(num_words))
```

→ **4**

```
print(1 in num_words)
```

→ **True**

```
print("one" in num_words)
```

→ **False**

(it is only checking keys, not vals)

```
for x in num_words:
    print(x, num_words[x])
```

→

```
2 two
1 one
0 zero
3 three
```

you can iterate over values
by combining a **for loop** with **lookup**

Extracting keys and values

```
num_words = {0:“zero”, 1:”one”, 2:”two”, 3:”three”}

print(type(num_words.keys()))

print(type(num_words.values()))
```

Extracting keys and values

```
num_words = {0:“zero”, 1:”one”, 2:”two”, 3:”three”}
```

```
print(type(num_words.keys()))    →  <class 'dict_keys'>

print(type(num_words.values()))  →  <class 'dict_values'>
```

don’t worry about these new types, because we can force them to be lists

Extracting keys and values

```
num_words = {0:“zero”, 1:”one”, 2:”two”, 3:”three”}
```

```
print(type(num_words.keys()))
```

→ **<class 'dict_keys'>**

```
print(type(num_words.values()))
```

→ **<class 'dict_values'>**

```
print(list(num_words.keys()))
```

→ **[3, 1, 2, 0]**

```
print(list(num_words.values()))
```

→ **[“one”, “two”, “zero”, “three”]**

Defaults with get and pop

```
suffix = {1:“st”, 2:”nd”, 3:”rd”}
```

Defaults with get and pop

```
suffix = {1:“st”, 2:”nd”, 3:”rd”}
```

```
✗ suffix.pop(0) # delete fails, because no key 0

✗ suffix[4] # lookup fails because no key 4
```

Defaults with get and pop

```
suffix = {1:“st”, 2:”nd”, 3:”rd”}

✗ suffix.pop(0) # delete fails, because no key 0

✗ suffix[4] # lookup fails because no key 4

✓ suffix.get(4, “th”) # returns “th” because no key 4
```

specify a default if key cannot be found

Defaults with get and pop

```
suffix = {1:“st”, 2:”nd”, 3:”rd”}
```

specify a default if
key cannot be found

```
✗ suffix.pop(0) # delete fails, because no key 0

✗ suffix[4] # lookup fails because no key 4

✓ suffix.get(4, “th”) # returns “th” because no key 4
```

specify a default if
key cannot be found

Defaults with get and pop

```
suffix = {1:“st”, 2:”nd”, 3:”rd”}
```

specify a default if
key cannot be found

```
✓ suffix.pop(0, “th”) # returns “th” because no key 0

✗ suffix[4] # lookup fails because no key 4

✓ suffix.get(4, “th”) # returns “th” because no key 4
```

specify a default if
key cannot be found

Defaults with get and pop

```
suffix = {1:“st”, 2:”nd”, 3:”rd”}


for num in range(6):
    print(str(num) + suffix.get(num, “th”))
```

Defaults with get and pop

```
suffix = {1:“st”, 2:”nd”, 3:”rd”}


for num in range(6):
    print(str(num) + suffix.get(num, “th”))
```

```
0th
1st
2nd
3rd
4th
5th
```

Today's Outline

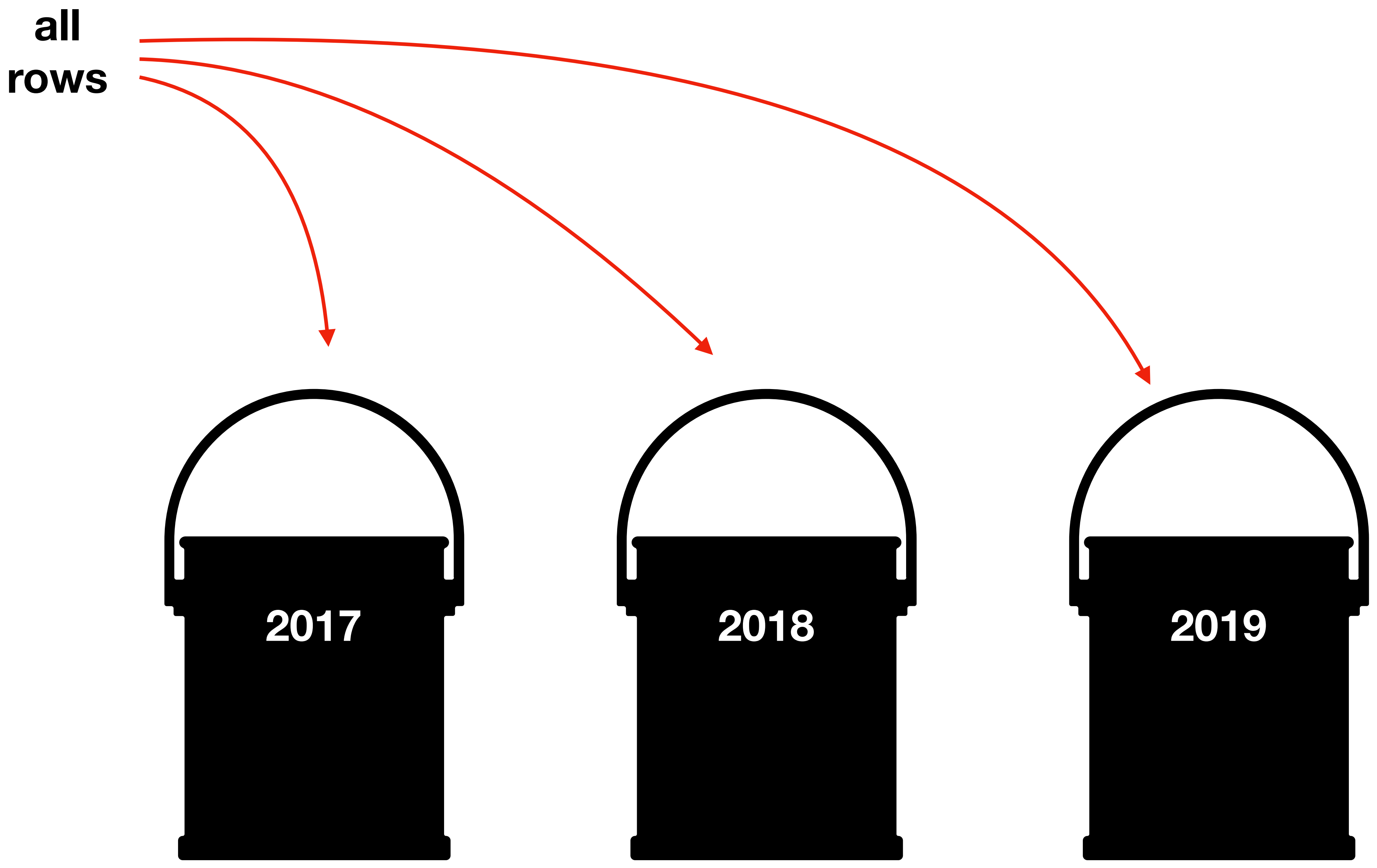
Dictionary Ops

Binning (dict of list)

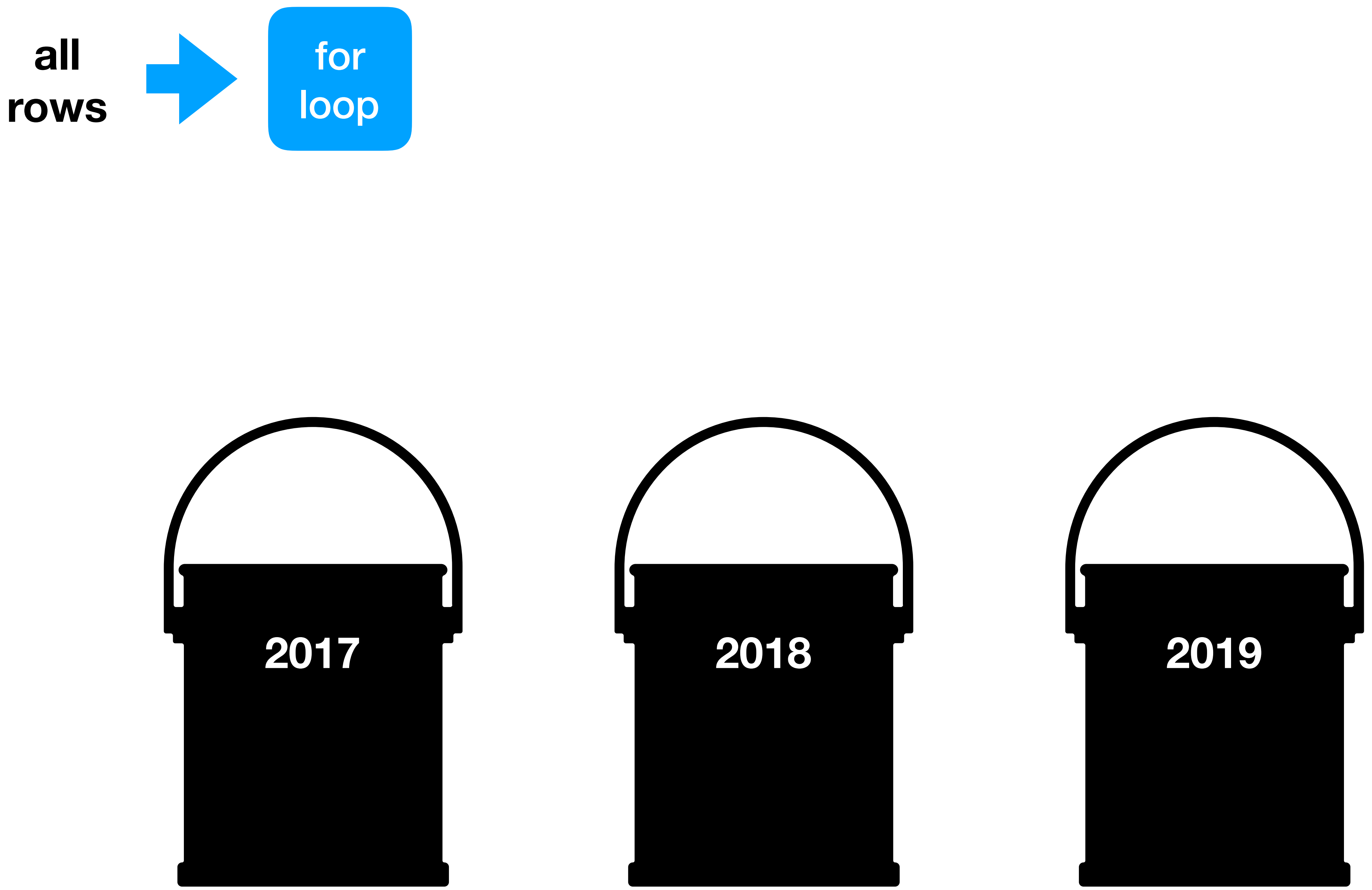
Table Representation (list of dict)

Markov Chains (dict of dict)

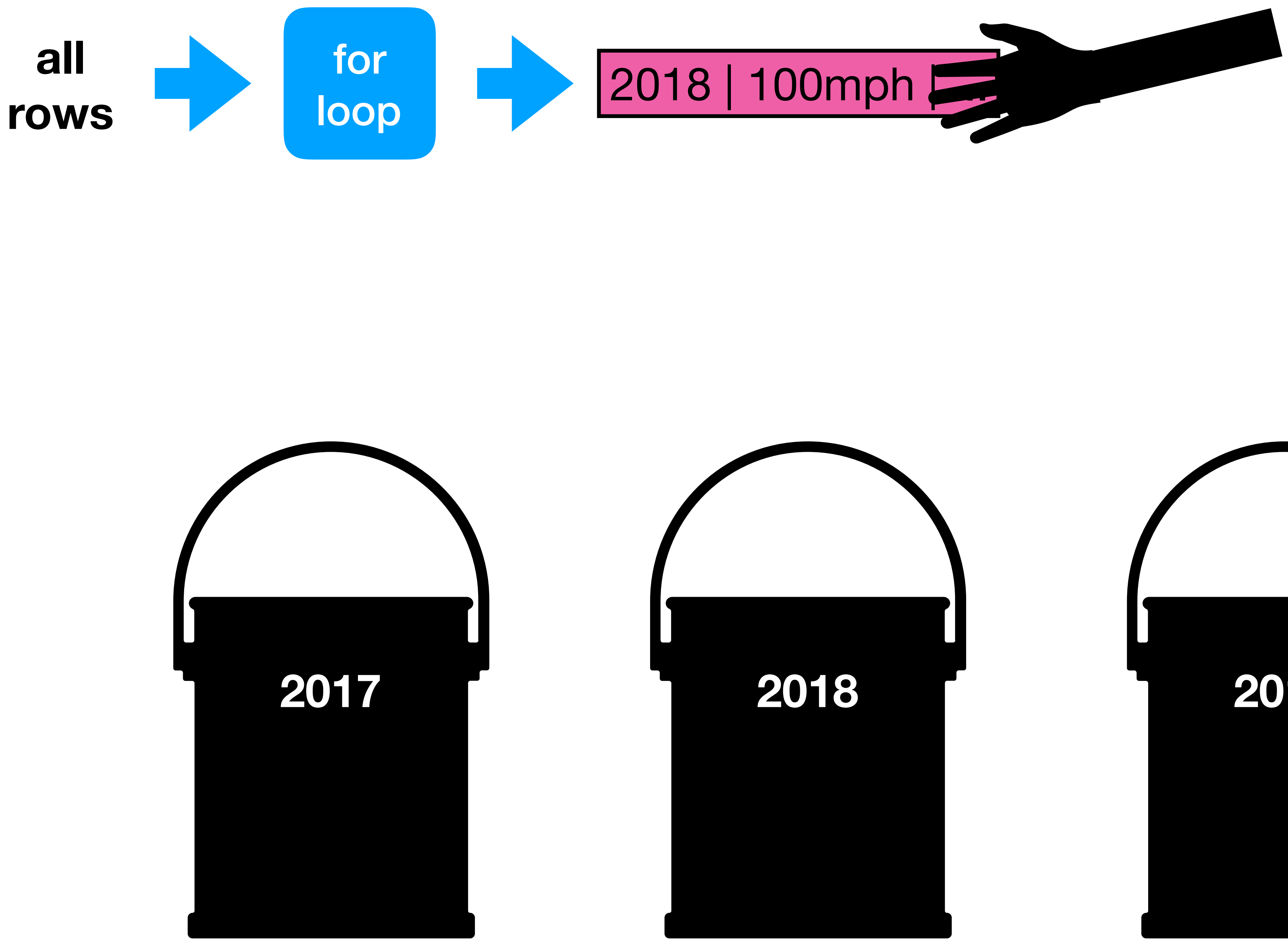
Bucketing/Binning



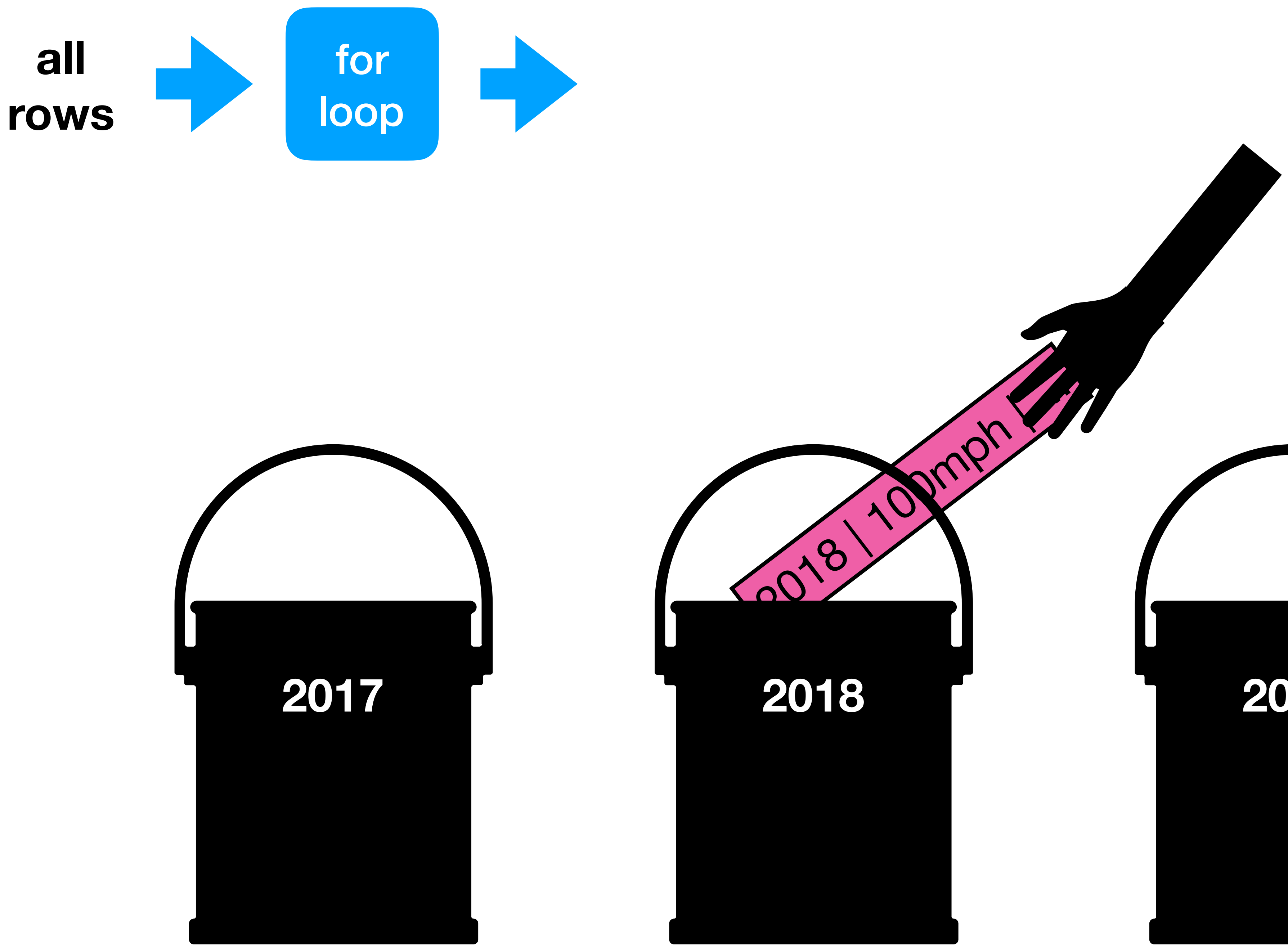
Bucketing/Binning



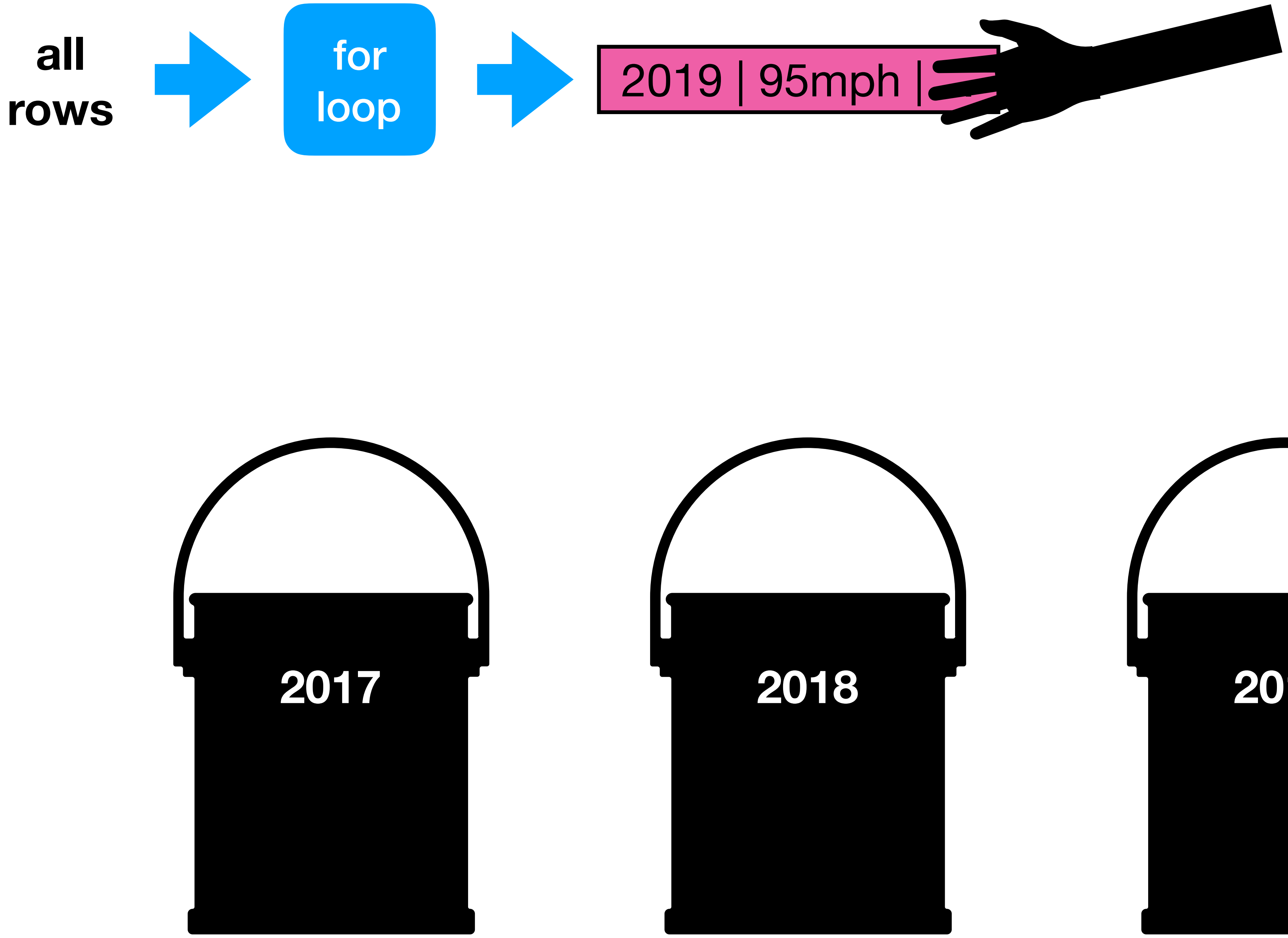
Bucketing/Binning



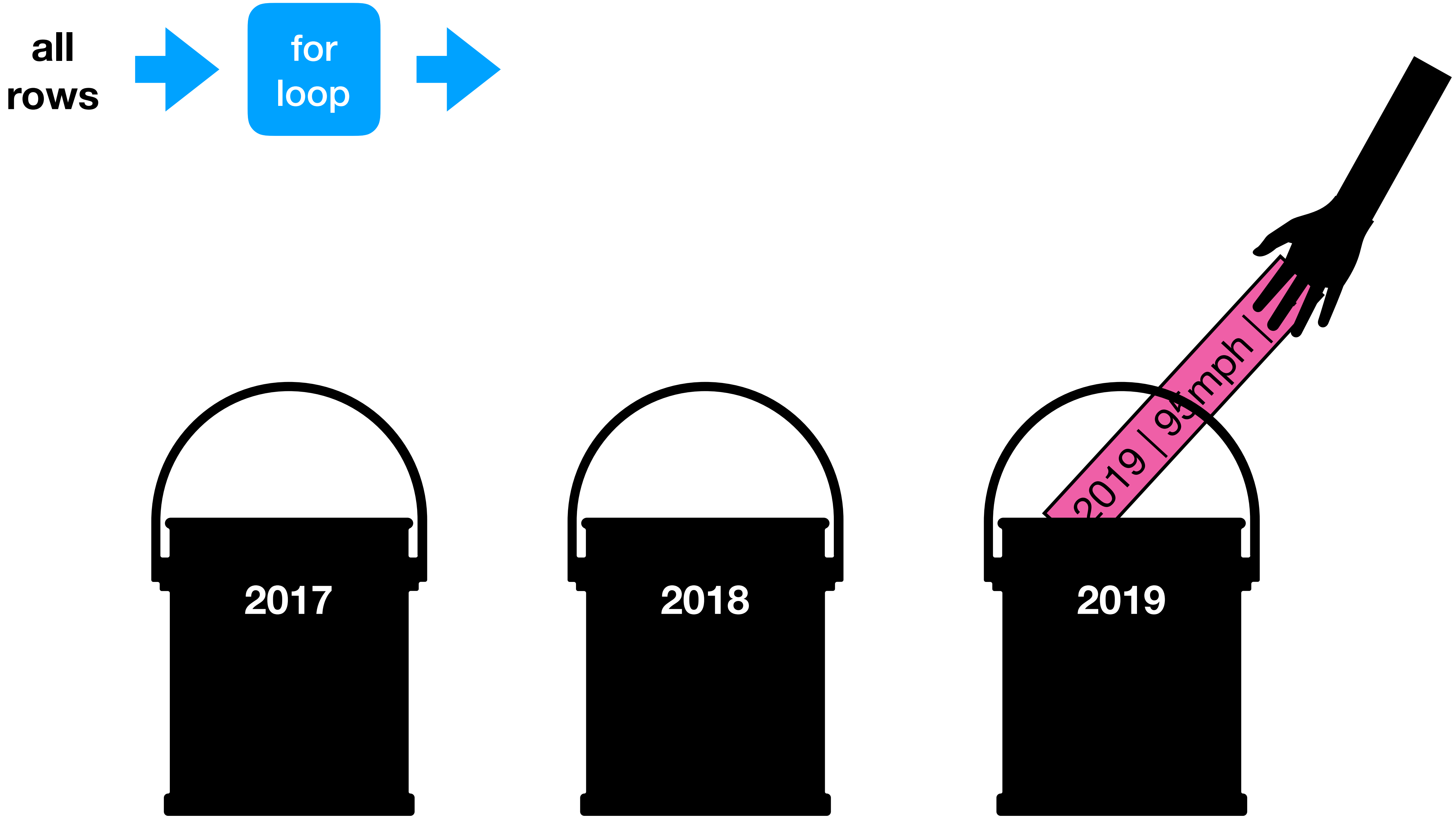
Bucketing/Binning



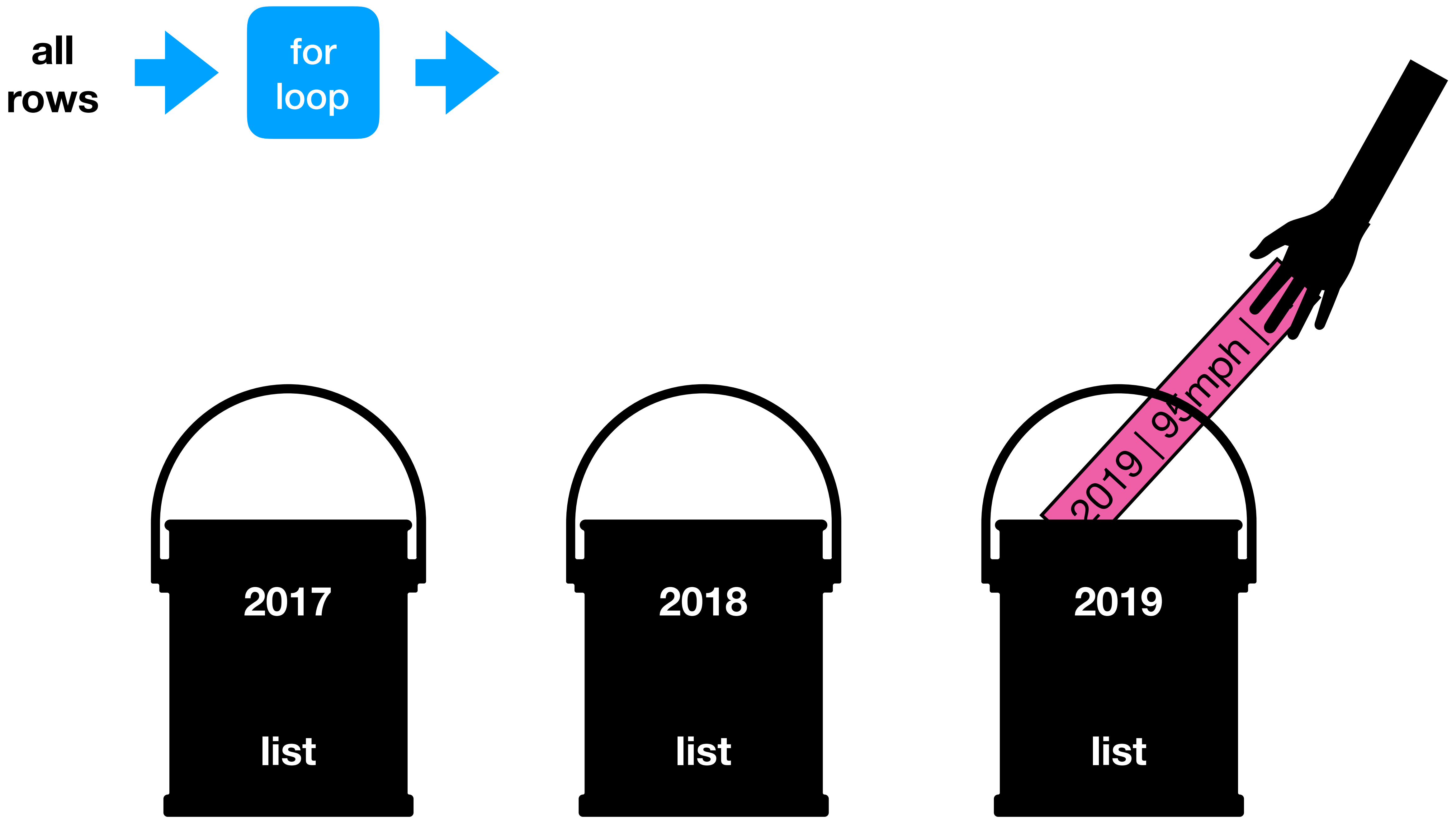
Bucketing/Binning



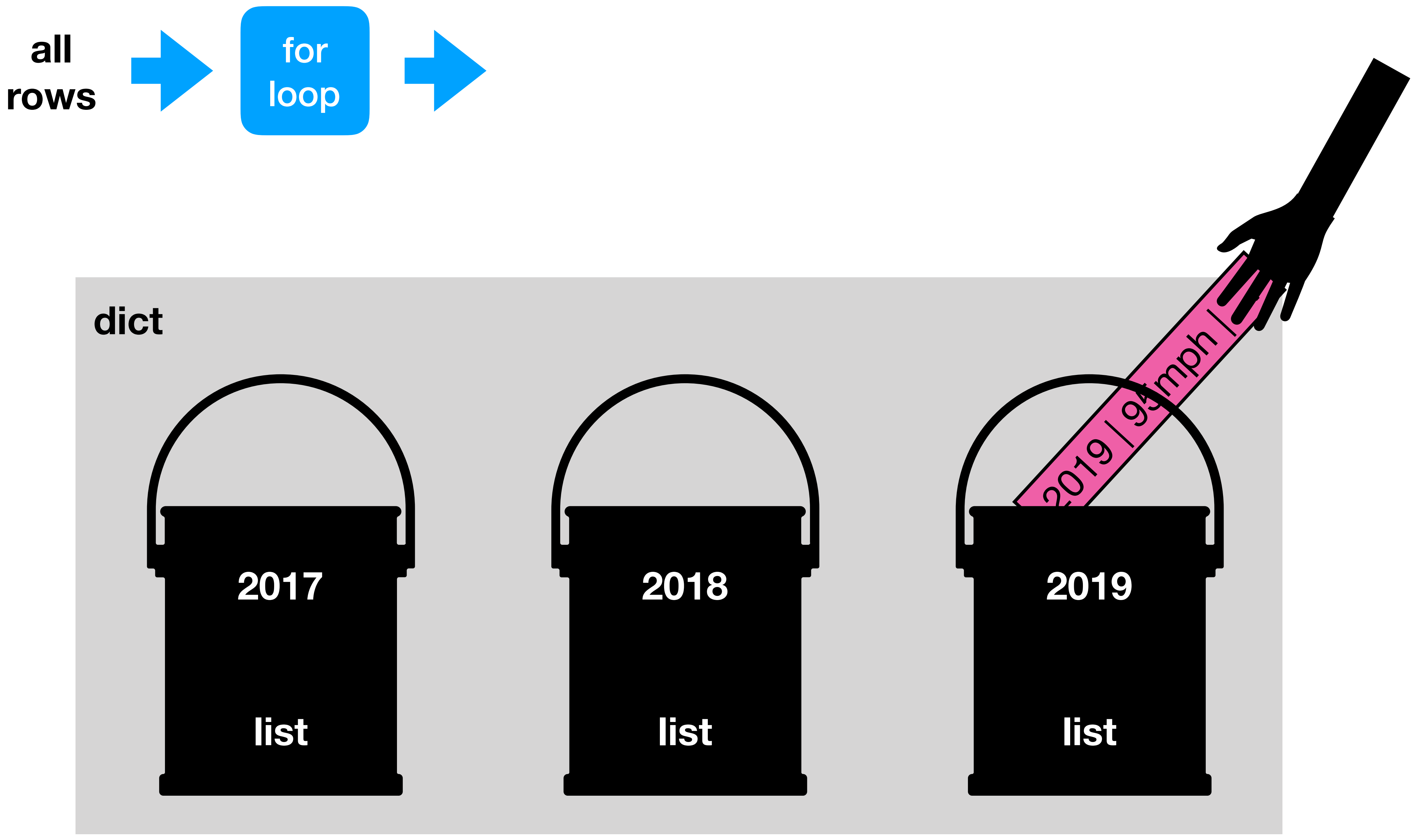
Bucketing/Binning



Bucketing/Binning



Bucketing/Binning



Bins with lists and dicts

all data

```
rows = [
    [2014, “A”, 123],
    [2015, “B”, 120],
    [2015, “C”, 140],
    [2016, “D”, 100],
    [2015, “E”, 130],
    [2016, “F”, 200],
]
```

Bins with lists and dicts

all data

```
rows = [
    [2014, “A”, 123],
    [2015, “B”, 120],
    [2015, “C”, 140],
    [2016, “D”, 100],
    [2015, “E”, 130],
    [2016, “F”, 200],
]
```

bin for 2014

```
bin2014 = [
    [2014, “A”, 123],
]
```

bin for 2015

```
bin2015 = [
    [2015, “B”, 120],
    [2015, “C”, 140],
    [2015, “E”, 130],
]
```

bin for 2016

```
bin2016 = [
    [2016, “D”, 100],
    [2016, “F”, 200],
]
```

Bins with lists and dicts

all data

```
rows = [
    [2014, “A”, 123],
    [2015, “B”, 120],
    [2015, “C”, 140],
    [2016, “D”, 100],
    [2015, “E”, 130],
    [2016, “F”, 200],
]
```

bin for 2014

```
bin2014 = [
    [2014, “A”, 123],
]
```

bin for 2015

```
bin2015 = [
    [2015, “B”, 120],
    [2015, “C”, 140],
    [2015, “E”, 130],
]
```

bin for 2016

```
bin2016 = [
    [2016, “D”, 100],
    [2016, “F”, 200],
]
```

how to keep track of all the lists?

Bins with lists and dicts

all data

```
rows = [
    [2014, “A”, 123],
    [2015, “B”, 120],
    [2015, “C”, 140],
    [2016, “D”, 100],
    [2015, “E”, 130],
    [2016, “F”, 200],
]
```

```
bins = {
    2014: [
        [2014, “A”, 123],
    ],
    2015: [
        [2015, “B”, 120],
        [2015, “C”, 140],
        [2015, “E”, 130],
    ],
    2016: [
        [2016, “D”, 100],
        [2016, “F”, 200],
    ]
}
```

Bins with lists and dicts

all data

```
rows = [
    [2014, “A”, 123],
    [2015, “B”, 120],
    [2015, “C”, 140],
    [2016, “D”, 100],
    [2015, “E”, 130],
    [2016, “F”, 200],
]
```

```
bins = {
    2014: [
        [2014, “A”, 123],
    ],
    2015: [
        [2015, “B”, 120],
        [2015, “C”, 140],
        [2015, “E”, 130],
    ],
    2016: [
        [2016, “D”, 100],
        [2016, “F”, 200],
    ]
}
```

Bins with lists and dicts

all data

```
rows = [
    [2014, “A”, 123],
    [2015, “B”, 120],
    [2015, “C”, 140],
    [2016, “D”, 100],
    [2015, “E”, 130],
    [2016, “F”, 200],
]
```

➡

```
bins = {
    2014: [
        [2014, “A”, 123],        ➡  median 123
    ],
    2015: [
        [2015, “B”, 120],
        [2015, “C”, 140],        ➡  median 130
        [2015, “E”, 130],
    ],
    2016: [
        [2016, “D”, 100],
        [2016, “F”, 200],        ➡  median 150
    ]
}
```

Demo 1: Median Tornado Speed per Year

Goal: print **median speed** of tornados for each year

Input:

- Tornado CSV

Output:

- Median within each year

Example:

```
prompt> python tornados.py
...
2015: 130
2016: 123
2017: 90
```

Today's Outline

Dictionary Ops

Binning (dict of list)

Table Representation (list of dict)

Markov Chains (dict of dict)

Table Representation

name	**x**	**y**
Alice	30	20
Bob	5	11
Cindy	-2	50

list of list representation

```
header = [“name”, “x”, “y”]
rows = [
    [“Alice”, 30, 20],
    [“Bob”,    5,  11],
    [“Cindy”, -2, 50],
]
```

Table Representation

name	x	y
Alice	30	20
Bob	5	11
Cindy	-2	50

list of list representation

list of dict representation

```
header = [“name”, “x”, “y”]
rows = [
    [“Alice”, 30, 20],
    [“Bob”,    5, 11],
    [“Cindy”, -2, 50],
]
```

```
[
  {“name”:“Alice”, “x”:30, “y”:20”},
  {“name”:“Bob”,   “x”:5,  “y”:11”},
  {“name”:“Cindy”, “x”:-2, “y”:50”},
]
```

Table Representation

name	x	y
Alice	30	20
Bob	5	11
Cindy	-2	$\boxed{50}$

list of list representation

list of dict representation

```
header = [“name”, “x”, “y”]
rows = [
    [“Alice”, 30, 20],
    [“Bob”,    5, 11],
    [“Cindy”, -2, 50],
]
```

```
[
  {“name”:“Alice”, “x”:30, “y”:20”},
  {“name”:“Bob”,   “x”:5,  “y”:11”},
  {“name”:“Cindy”, “x”:-2, “y”:50”},
]
```

Table Representation

name	x	y
Alice	30	20
Bob	5	11
Cindy	-2	50

list of list representation

list of dict representation

```
    header = [“name”, “x”, “y”]
    rows = [
        [“Alice”, 30, 20],
        [“Bob”,    5, 11],
2 →     [“Cindy”, -2, 50],
    ]
                      ↑
                      2
```

```
rows[2][header.index(“y”)]
```

```
[
  {“name”:“Alice”, “x”:30, “y”:20”},
  {“name”:“Bob”,   “x”:5,  “y”:11”},
  {“name”:“Cindy”, “x”:-2, “y”:50”},
]
```

Table Representation

name	x	y
Alice	30	20
Bob	5	11
Cindy	-2	50

list of list representation

```
header = ["name", "x", "y"]
rows = [
    ["Alice", 30, 20],
    ["Bob",    5, 11],
2 → ["Cindy", -2, 50],
]
                   ↑
                   2
```

`rows[2][header.index("y")]`

list of dict representation

```
[
    {"name":"Alice", "x":30, "y":20"},
    {"name":"Bob",   "x":5,  "y":11"},
2 → {"name":"Cindy", "x":-2, "y":50"},
]
                                  ↑
                                 "y"
```

`rows[2]["y"]`

Demo 2: Table Transform

Goal: create function that transforms list of lists table
to a list of dicts table

Input:

- List of lists (from a CSV)

Output:

- List of dicts

Example:

```
>>> header = [“x”,”y”]
>>> rows = [[1,2], [3,4]]
>>> transform(header, rows)
[{“x”:1, “y”:2}, {“x”:3, “y”:4}]
```

Today's Outline

Dictionary Ops

Binning (dict of list)

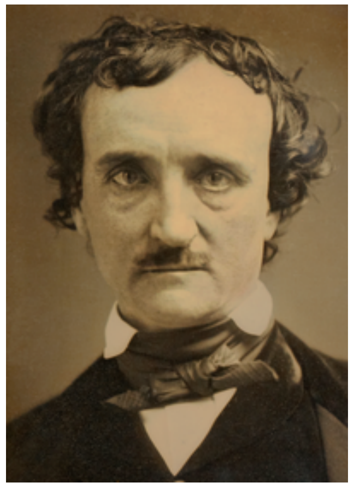
Table Representation (list of dict)

Markov Chains (dict of dict)

Demo 3: Letter Frequency

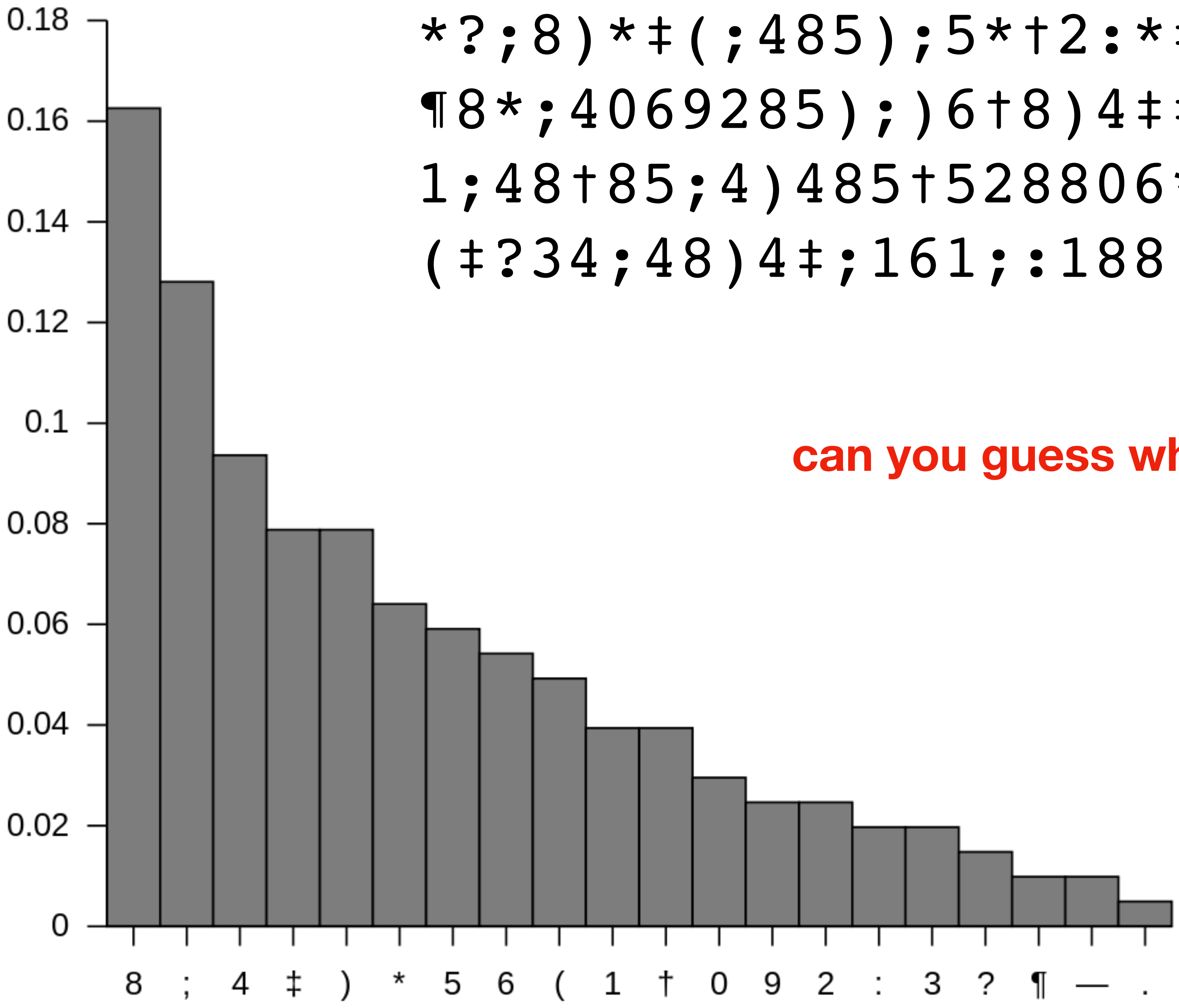
```
53‡‡†305))6*;4826)4‡.)4‡);806*;48†8
¶60))85;;]8*;:‡*8†83(88)5*†;46(;88*96
*?;8)*‡(;485);5*†2:*‡(;4956*2(5*—4)8
¶8*;4069285);)6†8)4‡‡;1(‡9;48081;8:8‡
1;48†85;4)485†528806*81(‡9;48;(88;4
(‡?34;48)4‡;161;:188;‡?;
```

can you guess what 8 represents?

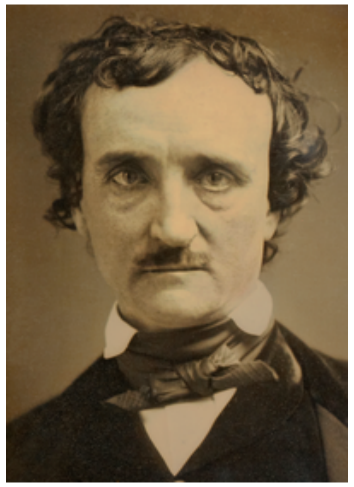


Demo 3: Letter Frequency

```
53‡‡†305))6*;4826)4‡.)4‡);806*;48†8
¶60))85;;]8*;:‡*8†83(88)5*†;46(;88*96
*?;8)*‡(;485);5*†2:*‡(;4956*2(5*—4)8
¶8*;4069285);)6†8)4‡‡;1(‡9;48081;8:8‡
1;48†85;4)485†528806*81(‡9;48;(88;4
(‡?34;48)4‡;161;:188;‡?;
```

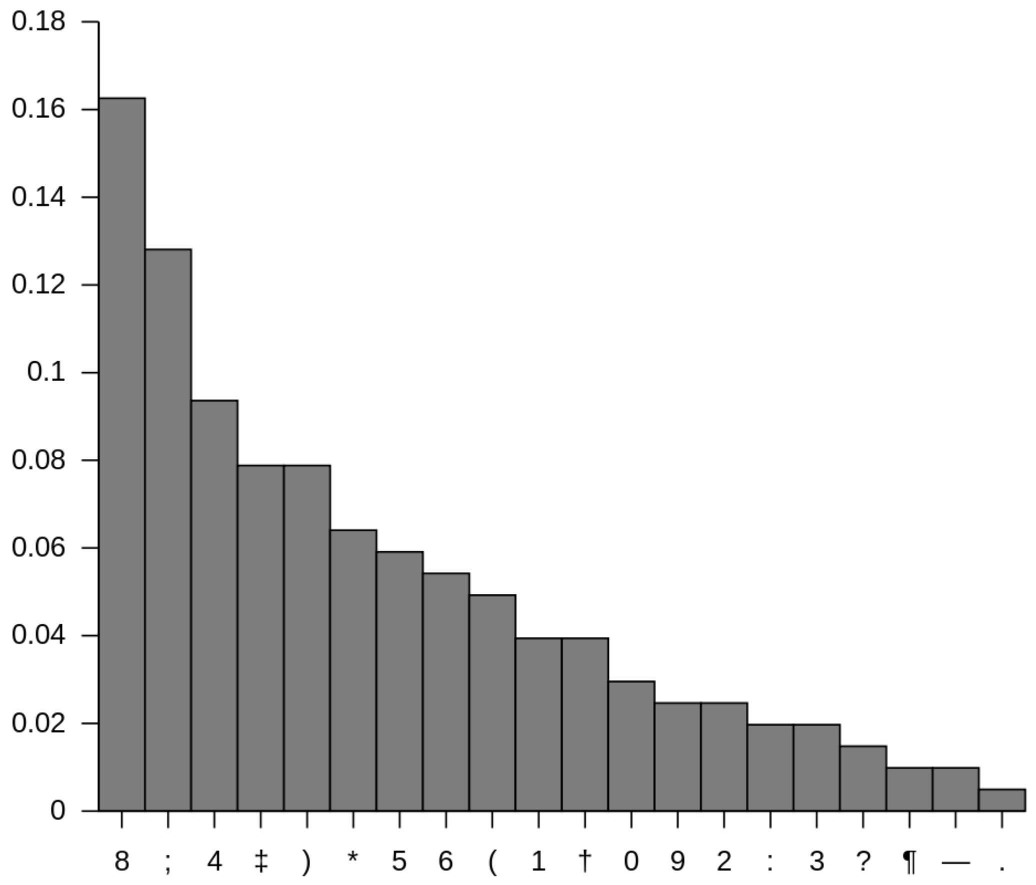


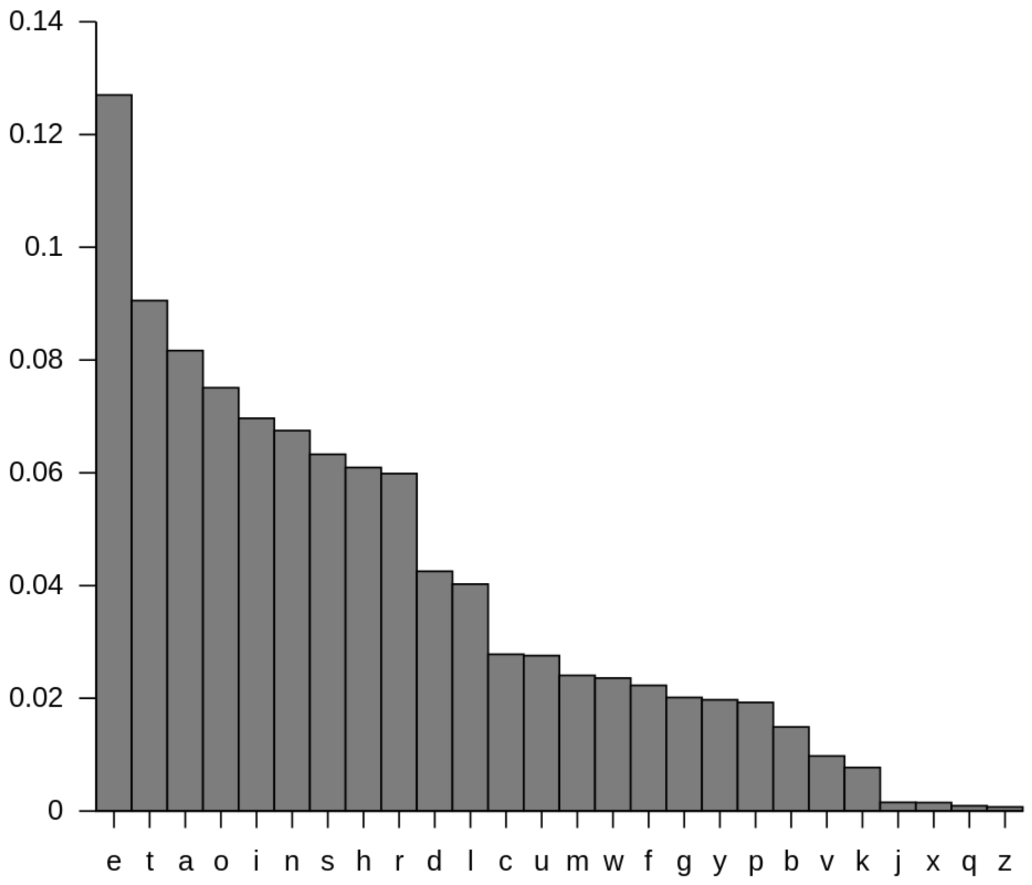
can you guess what 8 represents?



https://en.wikipedia.org/wiki/The_Gold-Bug

Demo 3: Letter Frequency

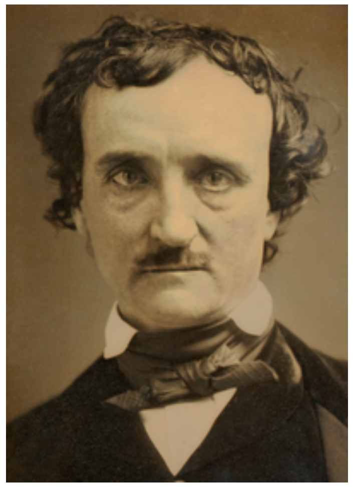




letters

symbols

how to compute these?



https://en.wikipedia.org/wiki/The_Gold-Bug

Demo 3: Letter Frequency

Goal: if we randomly pick a word in a text, what is the probability that it will be a given letter?

Input:

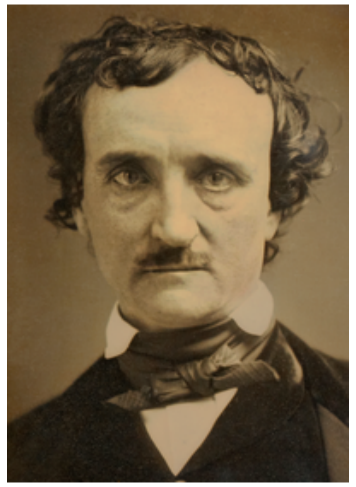
- Plaintext of book (from Project Gutenberg)

Output:

- The portion of letters in the text that are that letter

Example:

text: AAAAABBCCC
A: 50%
B: 20%
C: 30%


https://en.wikipedia.org/wiki/The_Gold-Bug

Sequence Data

Consider this sequence: “`the quick tiger is quiet`”

What letter likely comes after “t” in this text?

Sequence Data

Consider this sequence: “`the quick tiger is quiet`”

What letter likely comes after “t” in this text?

Next Letter	Probability
h	50%
i	50%
a	0%
...	0%

Sequence Data

Consider this sequence: “`the quick tiger is quiet`”

What letter likely comes after “t” in this text?

Next Letter	Probability
h	50%
i	50%
a	0%
…	0%

dict for “t”:

```
{“h”: 0.5, “i”: 0.5}
```

Sequence Data

Consider this sequence: `“the quick tiger is quiet”`

What letter likely comes after “t” in this text?

Next Letter	Probability
h	50%
i	50%
a	0%
...	0%

dict for “t”:

```
{“h”: 0.5, “i”: 0.5}
```

What letter likely comes after “q” in this text?

Sequence Data

Consider this sequence: `“the` **`qu`**`ick tiger is` **`qu`**`iet”`

What letter likely comes after “t” in this text?

Next Letter	Probability
h	50%
i	50%
a	0%
...	0%

dict for “t”:
`{“h”: 0.5, “i”: 0.5}`

What letter likely comes after “q” in this text?

Next Letter	Probability
u	100%
...	0%

dict for “q”:
`{“u”: 1.0}`

Sequence Data

Consider this sequence: “the quick tiger is quiet”

Imagine a next-letter probability dictionary for every letter

What letter likely comes after “t” in this text?

Next Letter	Probability
h	50%
i	50%
a	0%
…	0%

dict for “t”:

```
{“h”: 0.5, “i”: 0.5}
```

What letter likely comes after “q” in this text?

Next Letter	Probability
u	100%
…	0%

dict for “q”:

```
{“u”: 1.0}
```

Sequence Data

Consider this sequence: `“the quick tiger is quiet”`

What letter likely comes after “t” in this text?

Next Letter	Probability
h	50%
i	50%
a	0%
…	0%

What letter likely comes after “q” in this text?

Next Letter	Probability
u	100%
…	0%

Imagine a next-letter probability dictionary for every letter

dict for “u”:

```
{“i”: 1.0}
```

dict for “t”:

```
{“h”: 0.5, “i”: 0.5}
```

dict for “i”:

```
{“c”: 0.25, “g”: 0.25,
“s”: 0.25, “e”: 0.25}
```

dict for “q”:

```
{“u”: 1.0}
```

…

Sequence Data

Organize all the dicts with a dict:

```
probs = {
   "u":
}
```

Imagine a next-letter probability dictionary for every letter

dict for "u":

```
{"i": 1.0}
```

dict for "t":

```
{"h": 0.5, "i": 0.5}
```

dict for "i":

```
{"c": 0.25, "g": 0.25,
"s": 0.25, "e": 0.25}
```

dict for "q":

```
{"u": 1.0}
```

...

Sequence Data

Organize all the dicts with a dict:

```
probs = {
  “u”: {“i”: 1.0},



}
```

Imagine a next-letter probability dictionary for every letter

dict for “u”:

```
{“i”: 1.0}
```

dict for “t”:

```
{“h”: 0.5, “i”: 0.5}
```

dict for “i”:

```
{“c”: 0.25, “g”: 0.25,
“s”: 0.25, “e”: 0.25}
```

dict for “q”:

```
{“u”: 1.0}
```

...

Sequence Data

Organize all the dicts with a dict:

```
probs = {
  “u”: {“i”: 1.0},
  “t”: {“h”: 0.5, “i”: 0.5}
  “i”: {“c”: 0.25, “g”: 0.25,
        “s”: 0.25, “e”: 0.25},
  “q”: {“u”: 1.0},
  …
}
```

Imagine a next-letter probability dictionary for every letter

dict for “u”:

```
{“i”: 1.0}
```

dict for “t”:

```
{“h”: 0.5, “i”: 0.5}
```

dict for “i”:

```
{“c”: 0.25, “g”: 0.25,
“s”: 0.25, “e”: 0.25}
```

dict for “q”:

```
{“u”: 1.0}
```

…

Sequence Data

Organize all the dicts with a dict:

```
probs = {
   “u”: {“i”: 1.0},
   “t”: {“h”: 0.5, “i”: 0.5}
   “i”: {“c”: 0.25, “g”: 0.25,
         “s”: 0.25, “e”: 0.25},
   “q”: {“u”: 1.0},
   …
}
```

```
probs[“i”]
```

Imagine a next-letter probability dictionary for every letter

dict for “u”:

```
{“i”: 1.0}
```

dict for “t”:

```
{“h”: 0.5, “i”: 0.5}
```

dict for “i”:

```
{“c”: 0.25, “g”: 0.25,
“s”: 0.25, “e”: 0.25}
```

dict for “q”:

```
{“u”: 1.0}
```

...

Sequence Data

Organize all the dicts with a dict:

```
probs = {
   "u": {"i": 1.0},
   "t": {"h": 0.5, "i": 0.5}
   "i": {"c": 0.25, "g": 0.25,
         "s": 0.25, "e": 0.25},
   "q": {"u": 1.0},
   …
}
```

```
probs["i"]["e"] ➡ 0.25
```

There is a 25% probability that
the letter following an "i" is an "e"

Imagine a next-letter probability dictionary for every letter

dict for "u":

```
{"i": 1.0}
```

dict for "t":

```
{"h": 0.5, "i": 0.5}
```

dict for "i":

```
{"c": 0.25, "g": 0.25,
"s": 0.25, "e": 0.25}
```

dict for "q":

```
{"u": 1.0}
```

…

Vocabulary

```
probs = {
  “u”: {“i”: 1.0},
  “t”: {“h”: 0.5, “i”: 0.5}
  “i”: {“c”: 0.25, “g”: 0.25,
        “s”: 0.25, “e”: 0.25},
  “q”: {“u”: 1.0},
  …
}
```

The collection of transition probabilities like this is sometimes called a “stochastic matrix”

Processes that make probabilistic transitions like this (e.g., from one letter to the next) are called “Markov chains”

Random Text Generation

which looks closest to English?

1. XFOML RXKHRJFFJUJ ZLPWCFWKCYJ FFJEYVKCQSGHYD QPAAMKBZAACIBZLHJQD.

2. OCRO HLI RGWR NMIELWIS EU LL NBNESEBYA TH EEI ALHENHTTPA OOBTTVA NAH BRL.

3. ON IE ANTSOUTINYS ARE T INCTORE ST BE S DEAMY ACHIN D ILONASIVE TUCOOWE AT TEASONARE FUSO TIZIN ANDY TOBE SEACE CTISBE.

Examples from *A Mind at Play*, by Soni and Goodman

Random Text Generation

all letters equally likely

XFOML RXKHRJFFJUJ
ZLPWCFWKCYJ FFJEYVKCQSGHYD
QPAAMKBZAACIBZLHJQD.

weighted random, based on frequency in a text
(implement with dict)

OCRO HLI RGWR NMIELWIS EU LL
NBNESEBYA TH EEI ALHENHTTPA
OOBTTVA NAH BRL.

probability of each letter based on previous letter
(implement with dict of dicts)

ON IE ANTSOUTINYS ARE T
INCTORE ST BE S DEAMY ACHIN D
ILONASIVE TUCOOWE AT
TEASONARE FUSO TIZIN ANDY
TOBE SEACE CTISBE.

Examples from *A Mind at Play*, by Soni and Goodman

Random Text Generation

all letters equally likely

XFOML RXKHRJFFJUJ
ZLPWCFWKCYJ FFJEYVKCQSGHYD
QPAAMKBZAACIBZLHJQD.

weighted random, based on frequency in a text
(implement with dict)

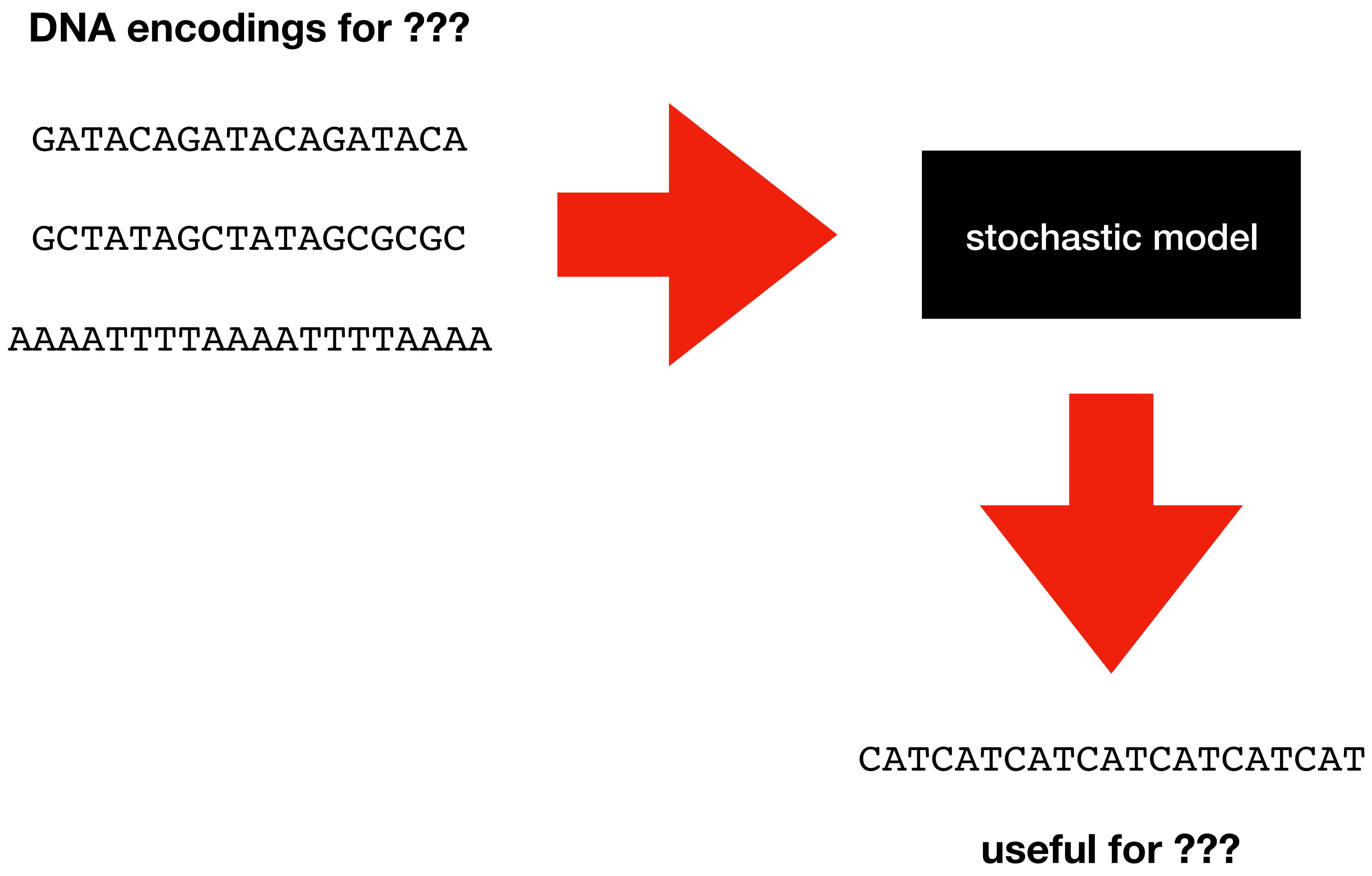
OCRO HLI RGWR NMIELWIS EU LL
NBNESEBYA TH EEI ALHENHTTPA
OOBTTVA NAH BRL.

probability of each letter based on previous letter
(implement with dict of dicts)

ON IE ANTSOUTINYS ARE T
INCTORE ST BE S DEAMY ACHIN D
ILONASIVE TUCOOWE AT
TEASONARE FUSO TIZIN ANDY
TOBE SEACE CTISBE.

Examples from *A Mind at Play*, by Soni and Goodman

Hypothetical Use Case



Demo 4: Conditional Letter Frequency

Goal: if we look at given letter,
what is the next letter likely to be?

Input:
- Plaintext of book (from Project Gutenberg)

Output:
- Transition probabilities
- Randomly generated text, based on probabilities

Demo 4: Conditional Letter Frequency

Goal: if we look at given letter,
what is the next letter likely to be?

Input:

- Plaintext of book (from Project Gutenberg)

Output:

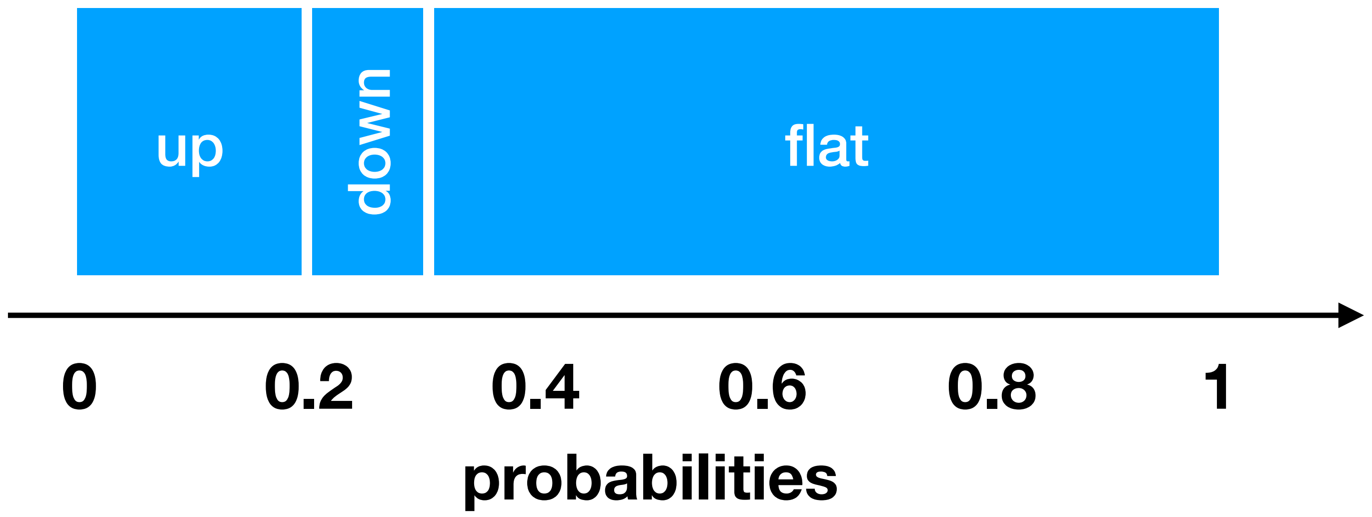
- Transition probabilities
- Randomly generated text, based on probabilities

Weighted Random

```
transitions = {
    “up”: 0.2,
    “down”: 0.1,
    “flat”: 0.7
}
```

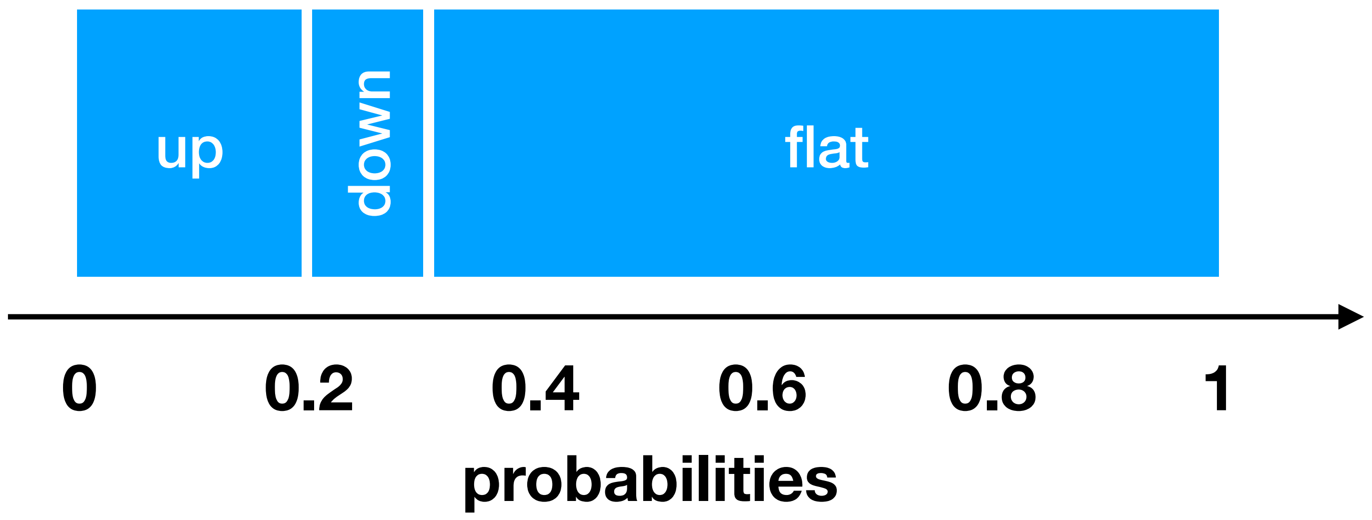
Weighted Random

```
transitions = {
    “up”: 0.2,
    “down”: 0.1,
    “flat”: 0.7
}
```



Weighted Random

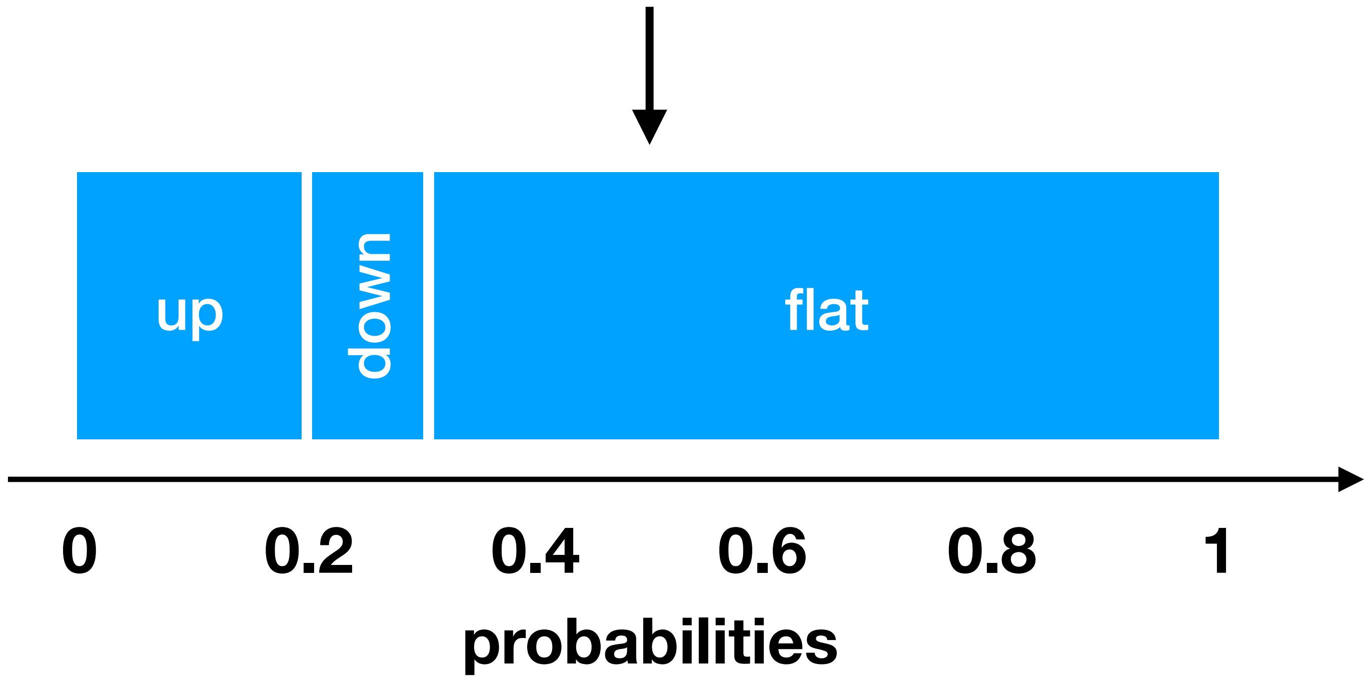
```
transitions = {
    “up”: 0.2,
    “down”: 0.1,
    “flat”: 0.7
}


x = random.random()
```



Weighted Random

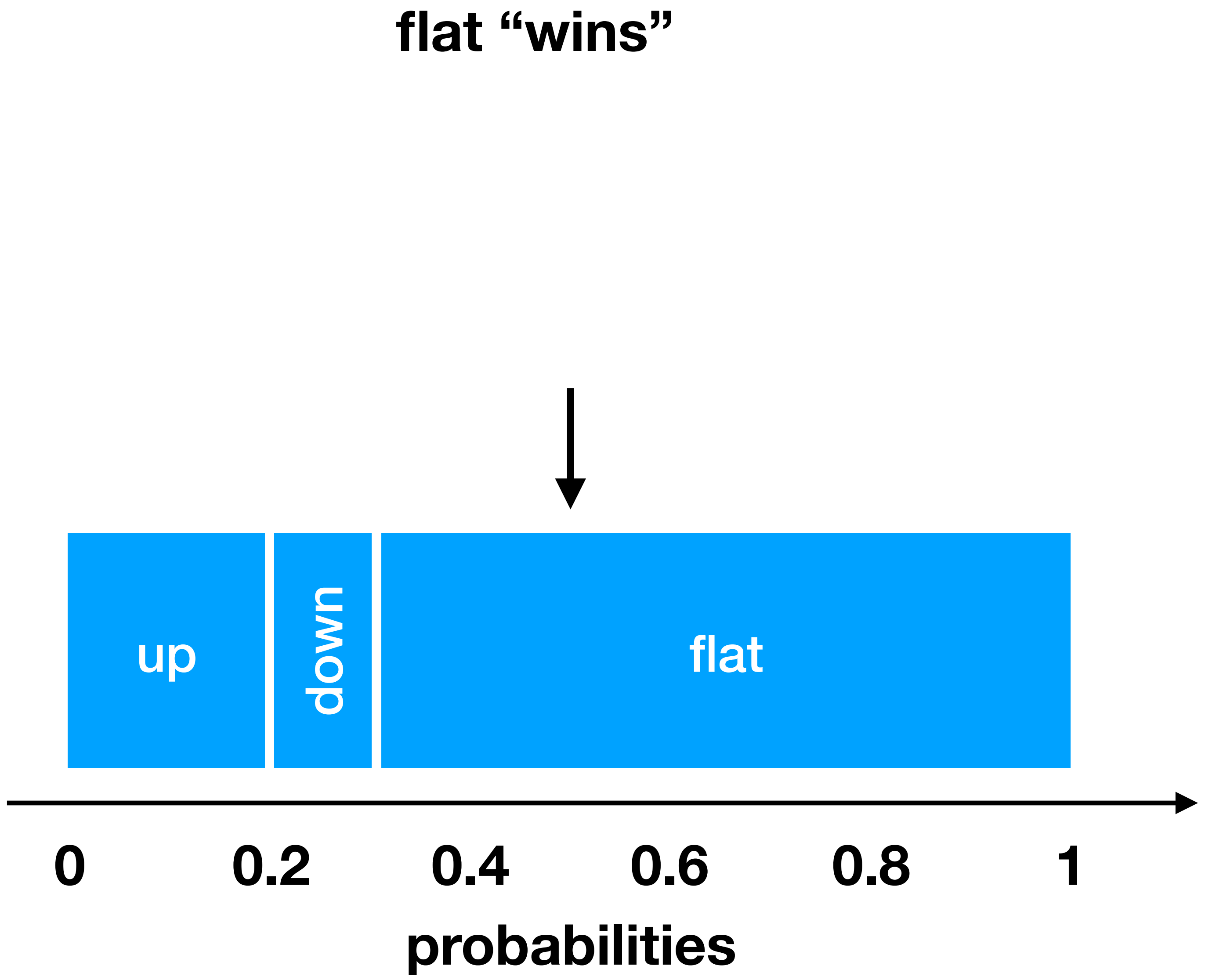
```
transitions = {
    “up”: 0.2,
    “down”: 0.1,
    “flat”: 0.7
}
```

```
x = random.random()
# assume 0.5
```



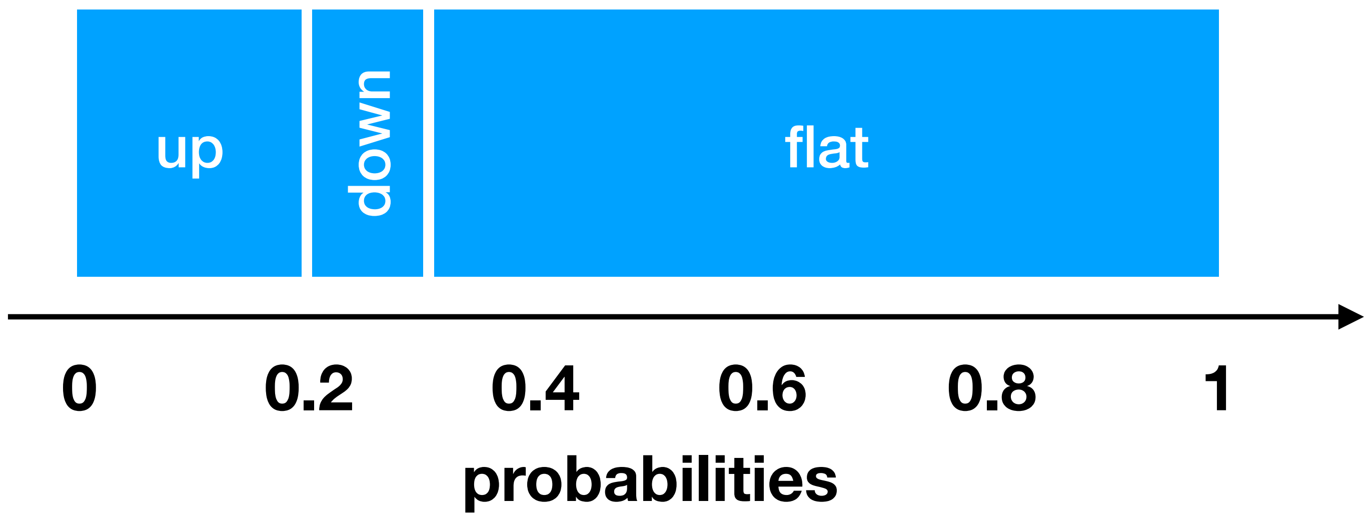
Weighted Random

```
transitions = {
    “up”: 0.2,
    “down”: 0.1,
    “flat”: 0.7
}


x = random.random()
# assume 0.5
```



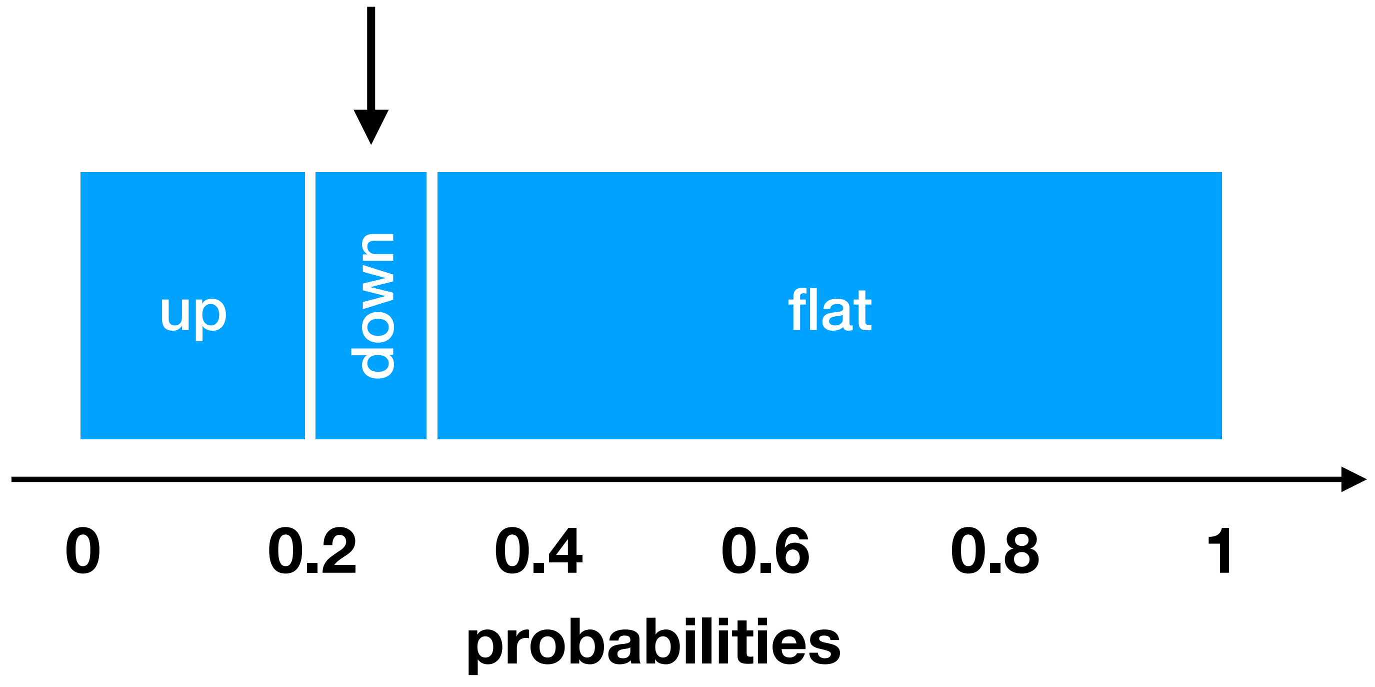
Weighted Random

```
transitions = {
    “up”: 0.2,
    “down”: 0.1,
    “flat”: 0.7
}

x = random.random()
# assume 0.25
```



Weighted Random

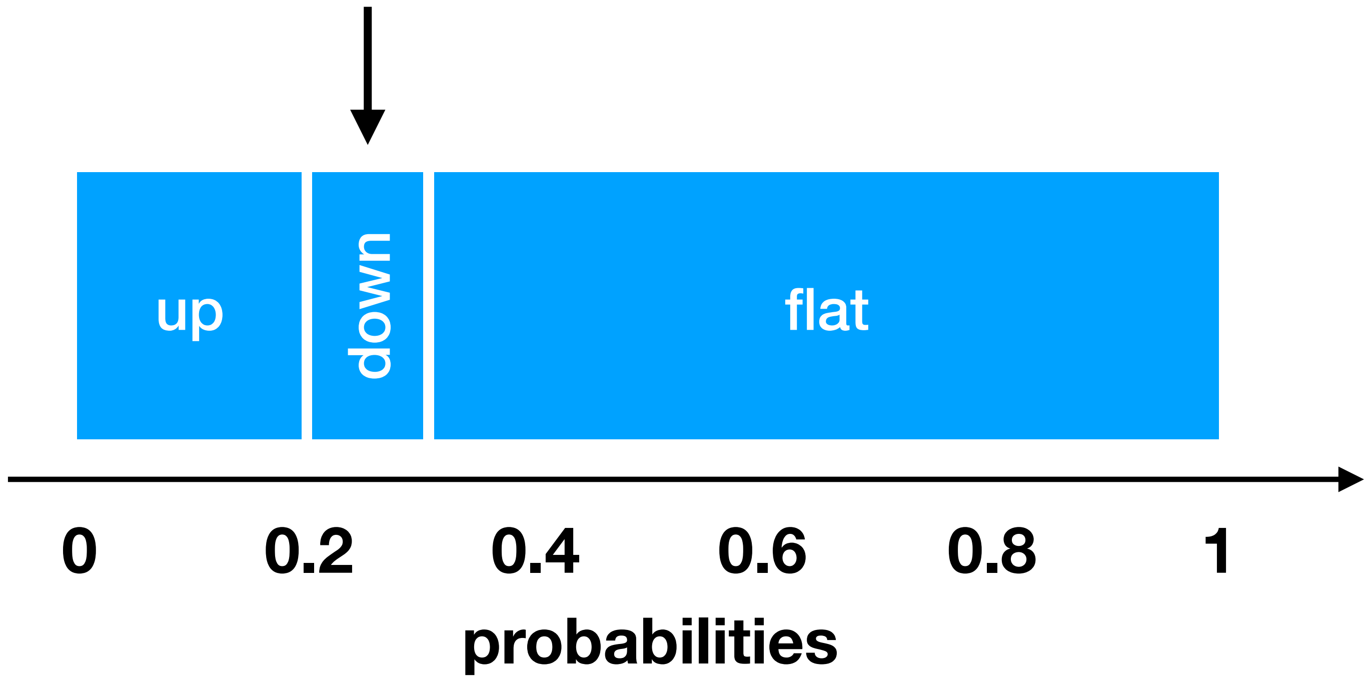
```
transitions = {
    “up”: 0.2,
    “down”: 0.1,
    “flat”: 0.7
}

x = random.random()
# assume 0.25
```



Weighted Random

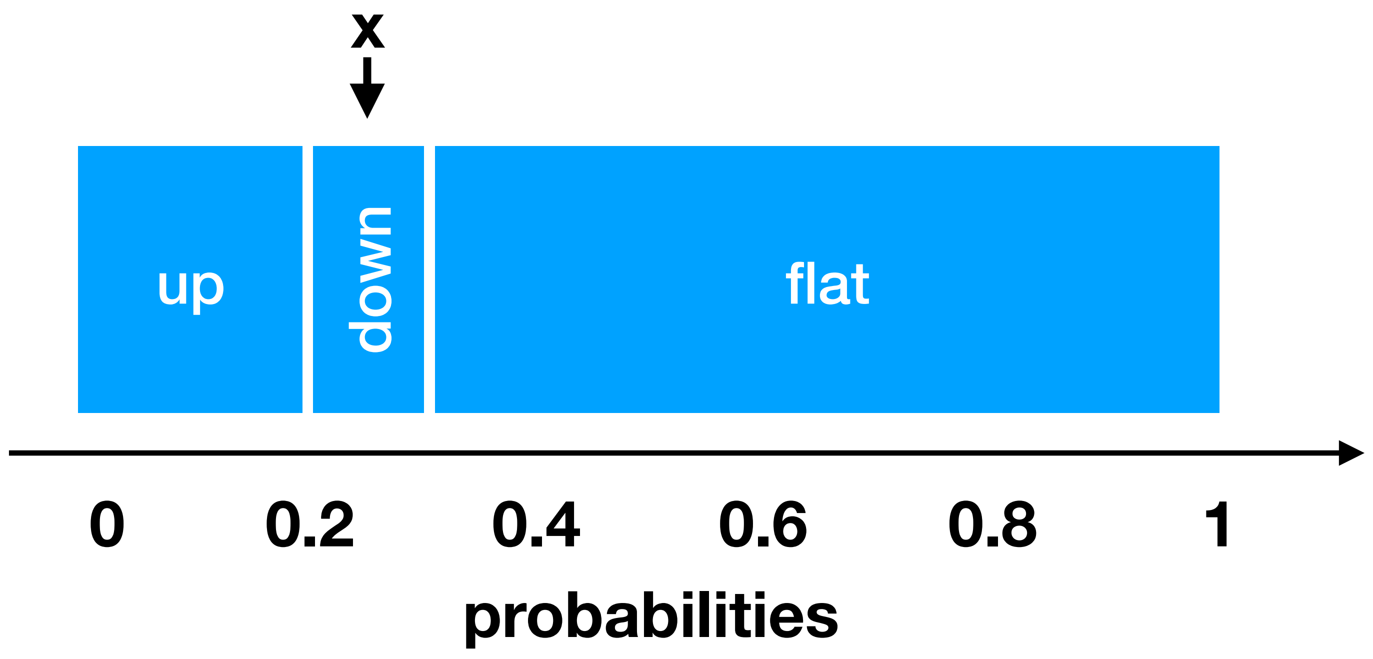
```
transitions = {
    “up”: 0.2,
    “down”: 0.1,
    “flat”: 0.7
}

x = random.random()
# assume 0.25
```

down “wins”



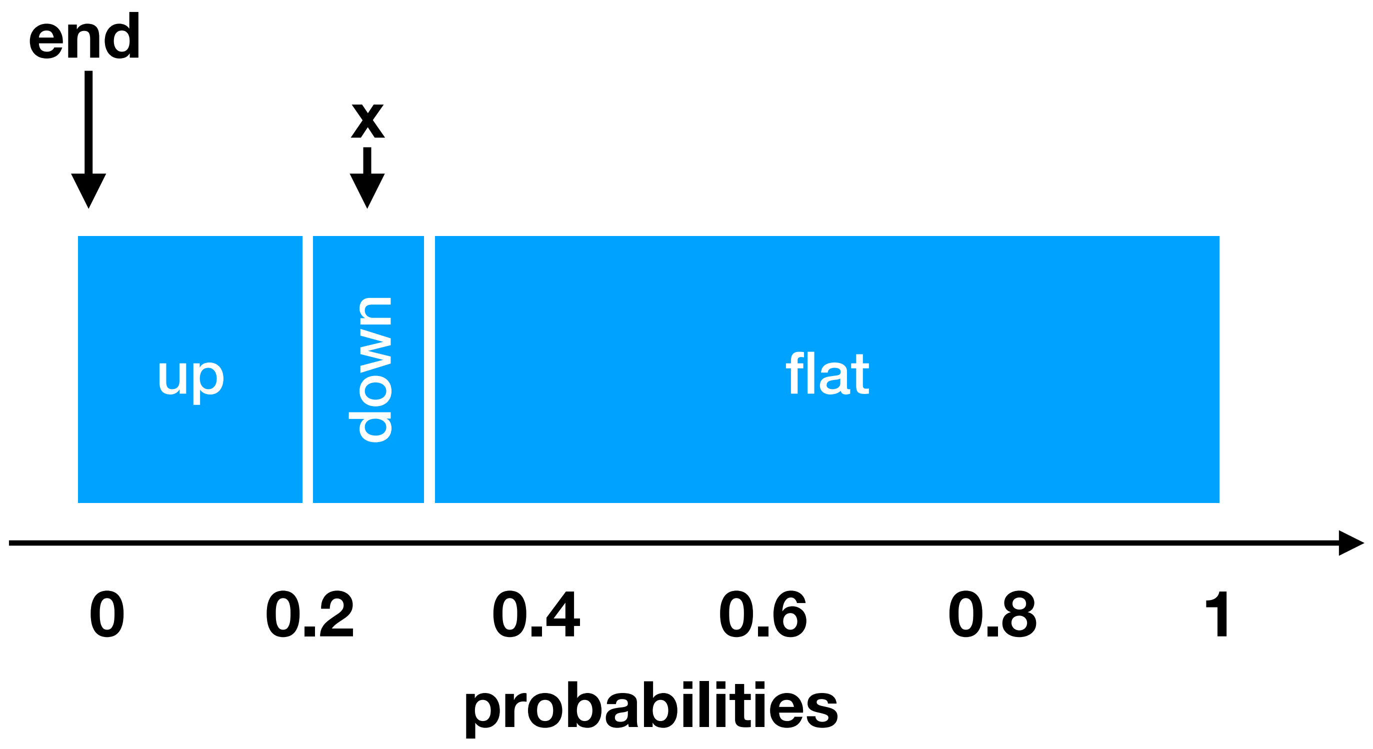
Weighted Random

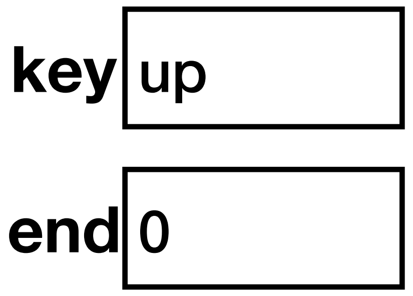
```
transitions = {
    “up”: 0.2,
    “down”: 0.1,
    “flat”: 0.7
}


x = random.random()
# assume 0.25


end = 0
keys = [“up”, “down”, “flat”]
winner = None
for key in keys:
    end += transitions[key]
    if end >= x:
        winner = key
        break
```



Weighted Random

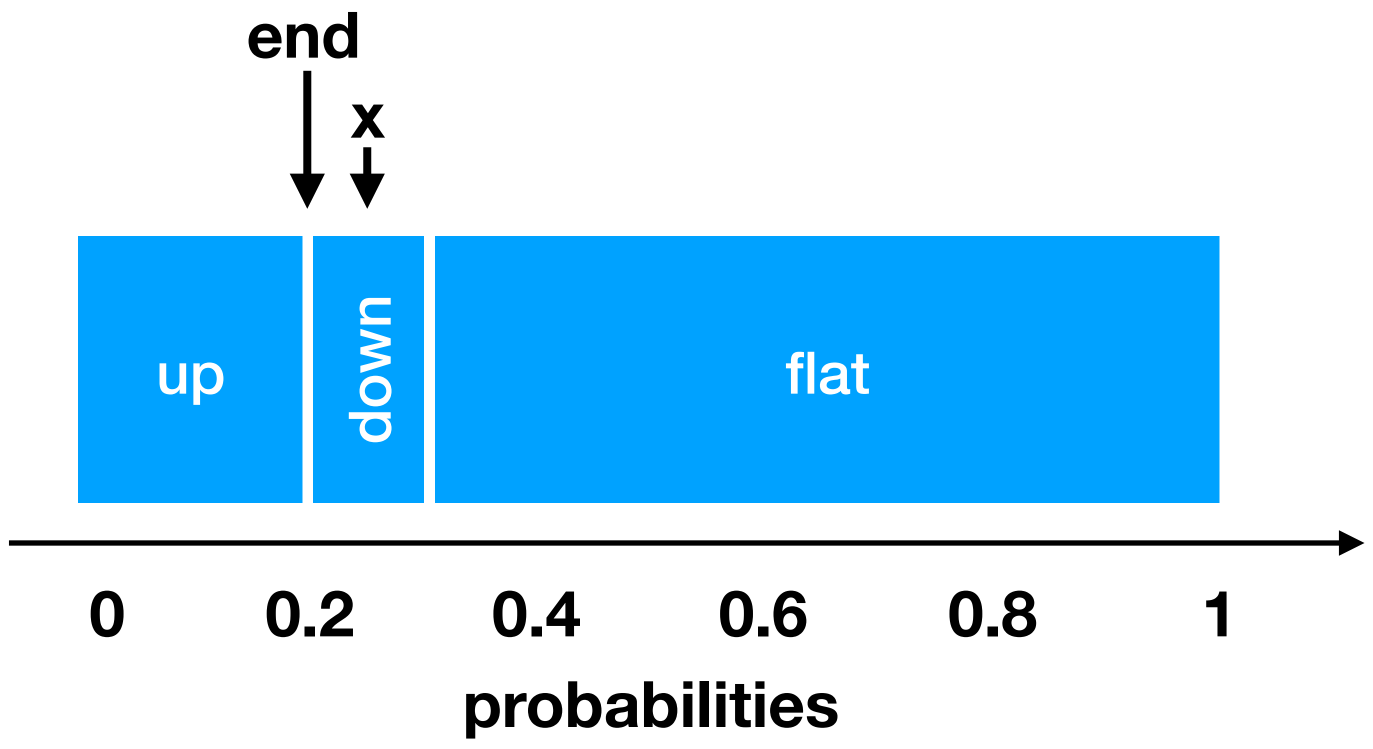
```
transitions = {
    “up”: 0.2,
    “down”: 0.1,
    “flat”: 0.7
}


x = random.random()
# assume 0.25


end = 0
keys = [“up”, “down”, “flat”]
winner = None
for key in keys:
    end += transitions[key]
    if end >= x:
        winner = key
        break
```

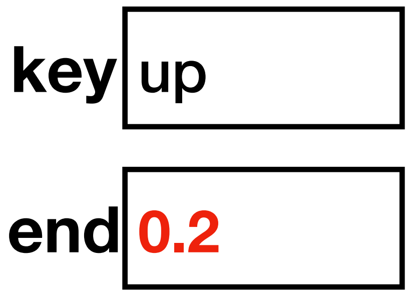




Weighted Random

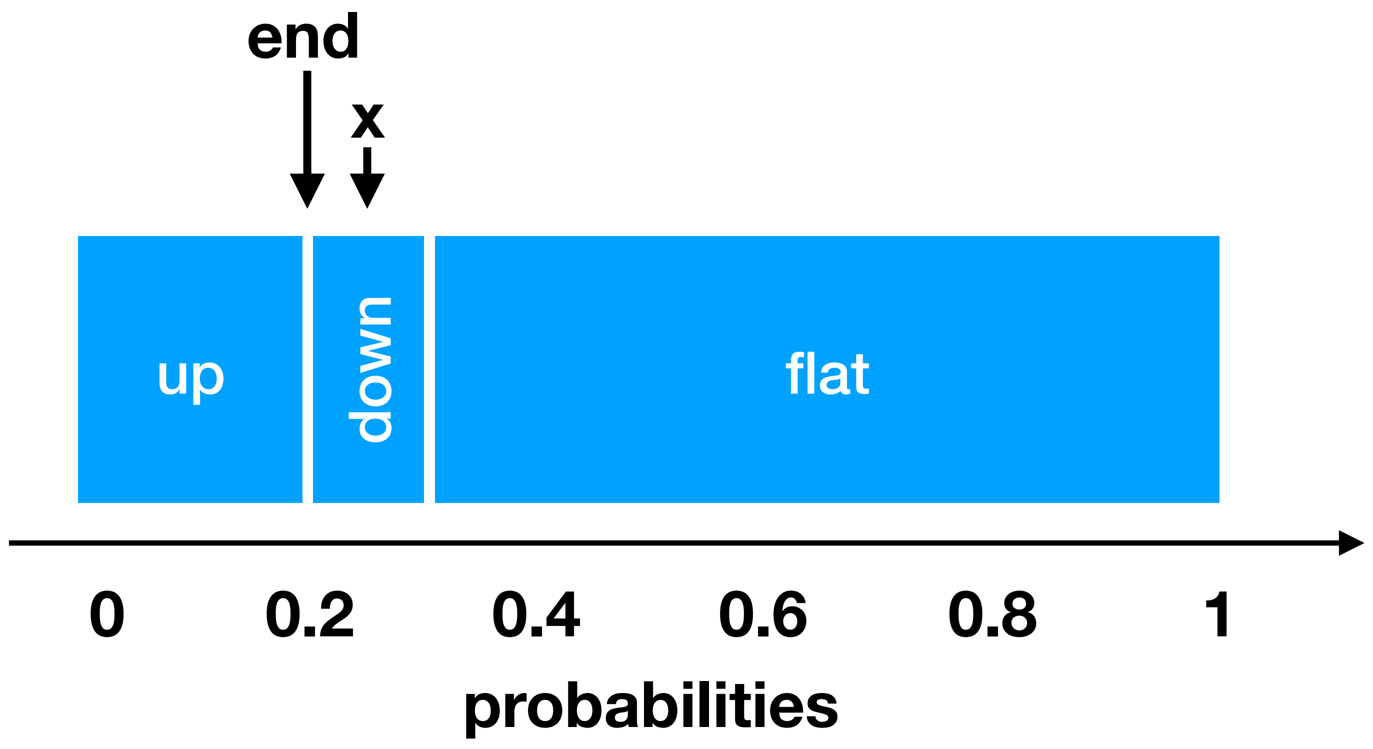
```
transitions = {
    “up”: 0.2,
    “down”: 0.1,
    “flat”: 0.7
}

x = random.random()
# assume 0.25

end = 0
keys = [“up”, “down”, “flat”]
winner = None
for key in keys:
    end += transitions[key]
    if end >= x:
        winner = key
        break
```

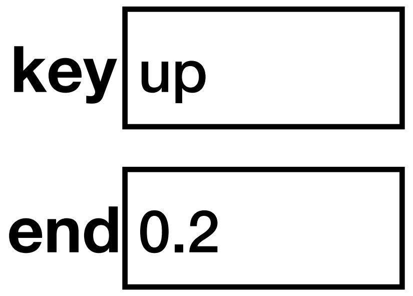




Weighted Random

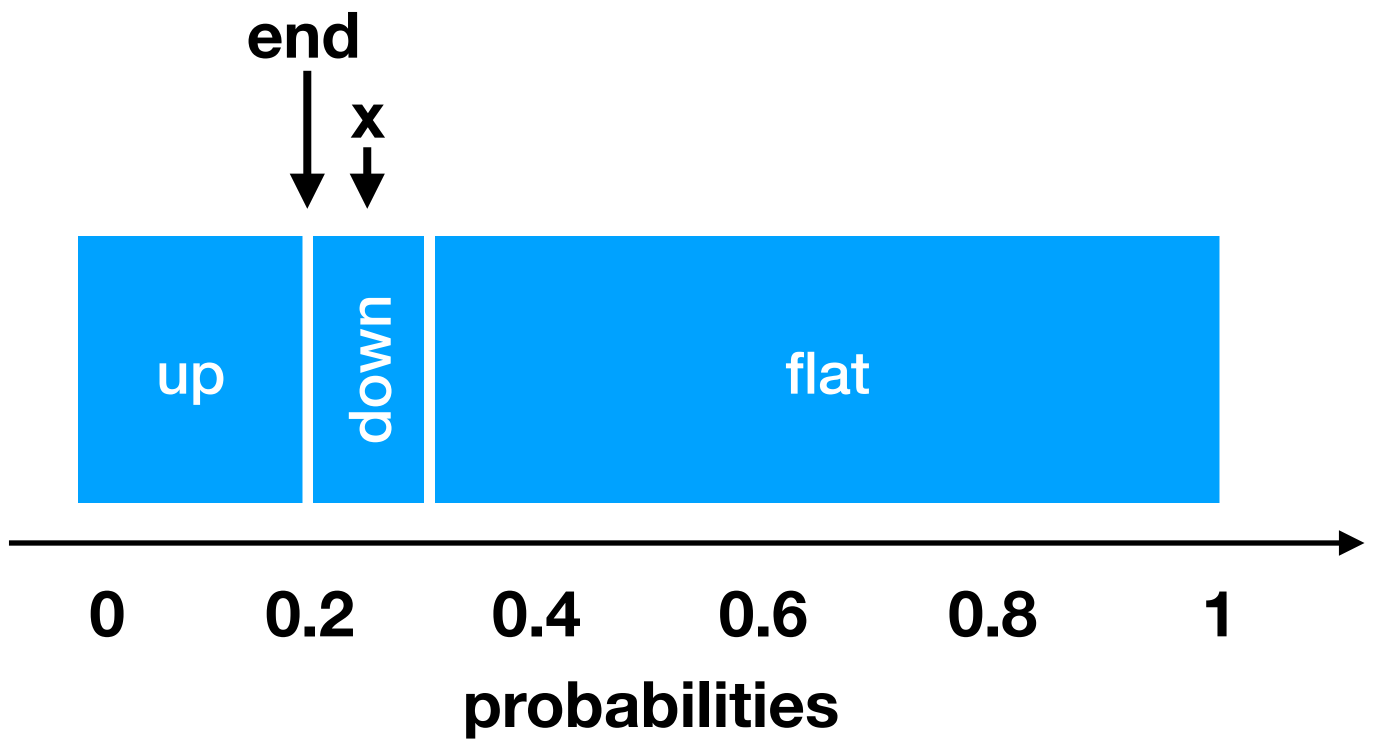
```
transitions = {
    “up”: 0.2,
    “down”: 0.1,
    “flat”: 0.7
}


x = random.random()
# assume 0.25


end = 0
keys = [“up”, “down”, “flat”]
winner = None
for key in keys:
    end += transitions[key]
    if end >= x:
        winner = key
        break
```





Weighted Random

```
transitions = {
    “up”: 0.2,
    “down”: 0.1,
    “flat”: 0.7
}

x = random.random()
# assume 0.25

end = 0
keys = [“up”, “down”, “flat”]
winner = None
for key in keys:
    end += transitions[key]
    if end >= x:
        winner = key
        break
```



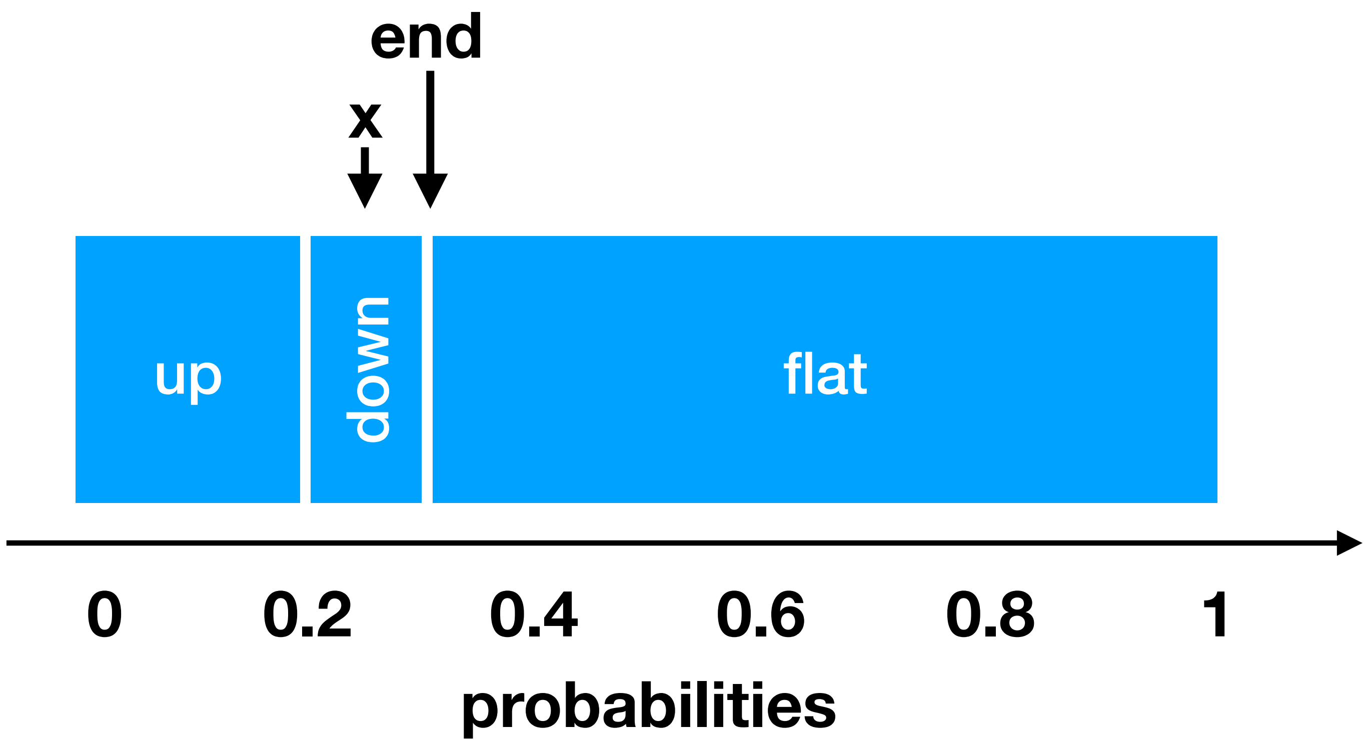
key	down
end	0.2

Weighted Random

```
transitions = {
    “up”: 0.2,
    “down”: 0.1,
    “flat”: 0.7
}
```

```
x = random.random()
# assume 0.25
```

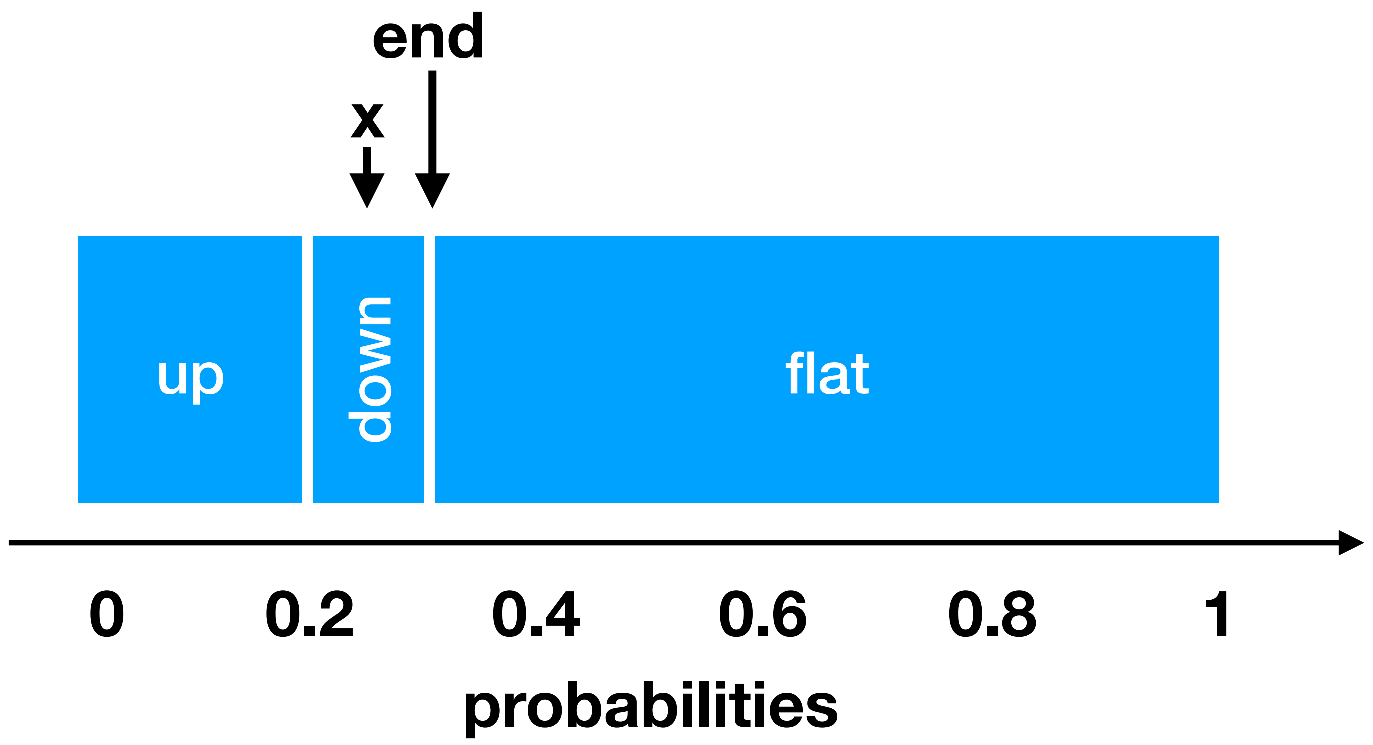
```
end = 0
keys = [“up”, “down”, “flat”]
winner = None
for key in keys:
    end += transitions[key]
    if end >= x:
        winner = key
        break
```



key	down
end	0.3

Weighted Random

```
transitions = {
    “up”: 0.2,
    “down”: 0.1,
    “flat”: 0.7
}


x = random.random()
# assume 0.25


end = 0
keys = [“up”, “down”, “flat”]
winner = None
for key in keys:
    end += transitions[key]
    if end >= x:
        winner = key
        break
```



key: down

end: 0.3

we randomly chose “down”