

Find the Pattern

Efficient
Phrases

UNCONSCIOUS OMISSION

COLD WINDOWSILL

INSIDIOUS DOMINION

VOLUMINOUS PILLOWS

VIVID DISILLUSIONS

Data Structures

January 15, 2019

Welcome Back!

Recall

Efficient

Phrases

UNCONSCIOUS OMISSION

COLD WINDOWSILL

INSIDIOUS DOMINION

VOLUMINOUS PILLOWS

VIVID DISILLUSIONS

Recall

Efficient
Phrases

UNCONSCIOUS OMISSION

COLD WINDOWSILL

INSIDIOUS DOMINION

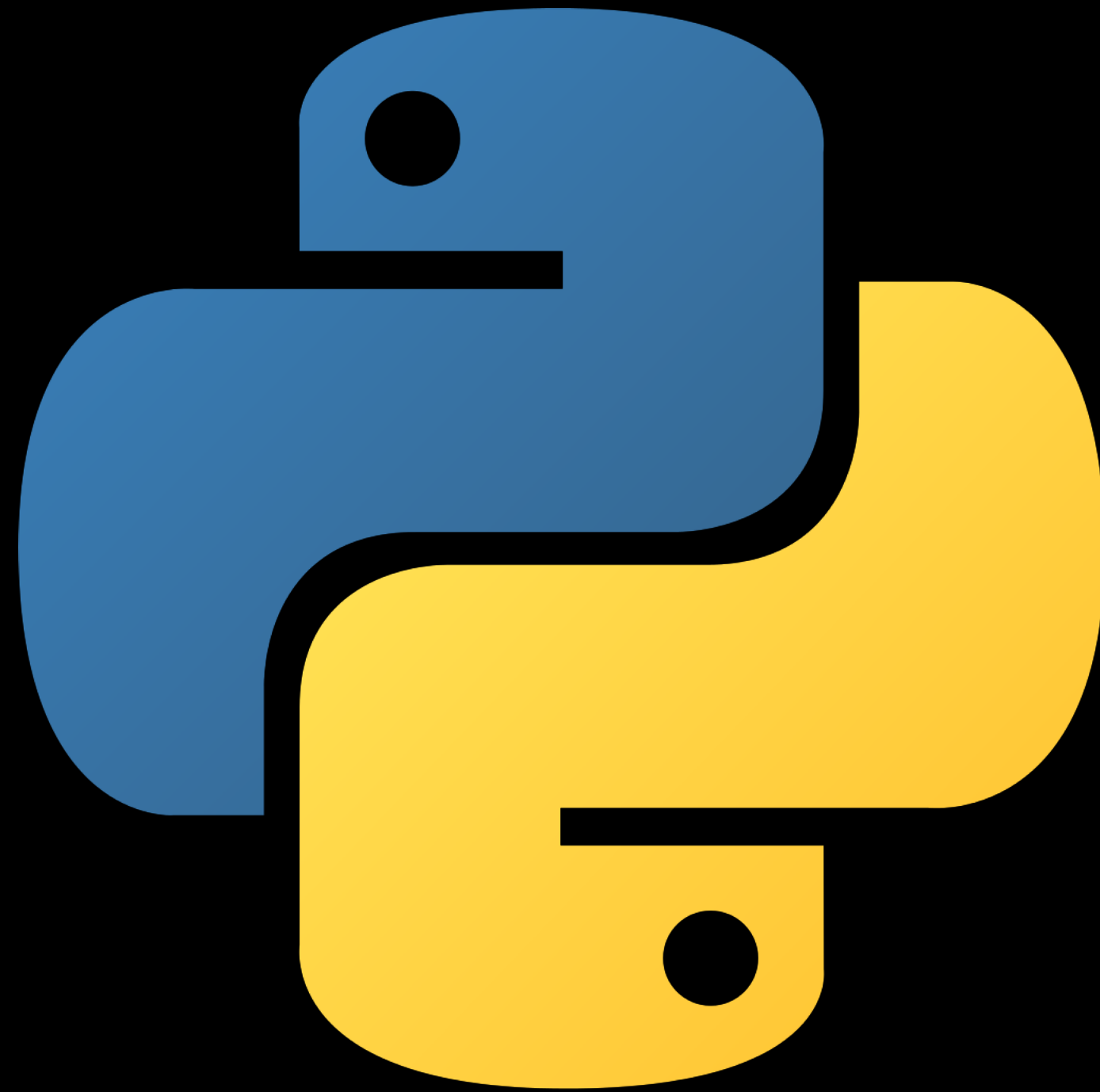
VOLUMINOUS PILLOWS

VIVID DISILLUSIONS

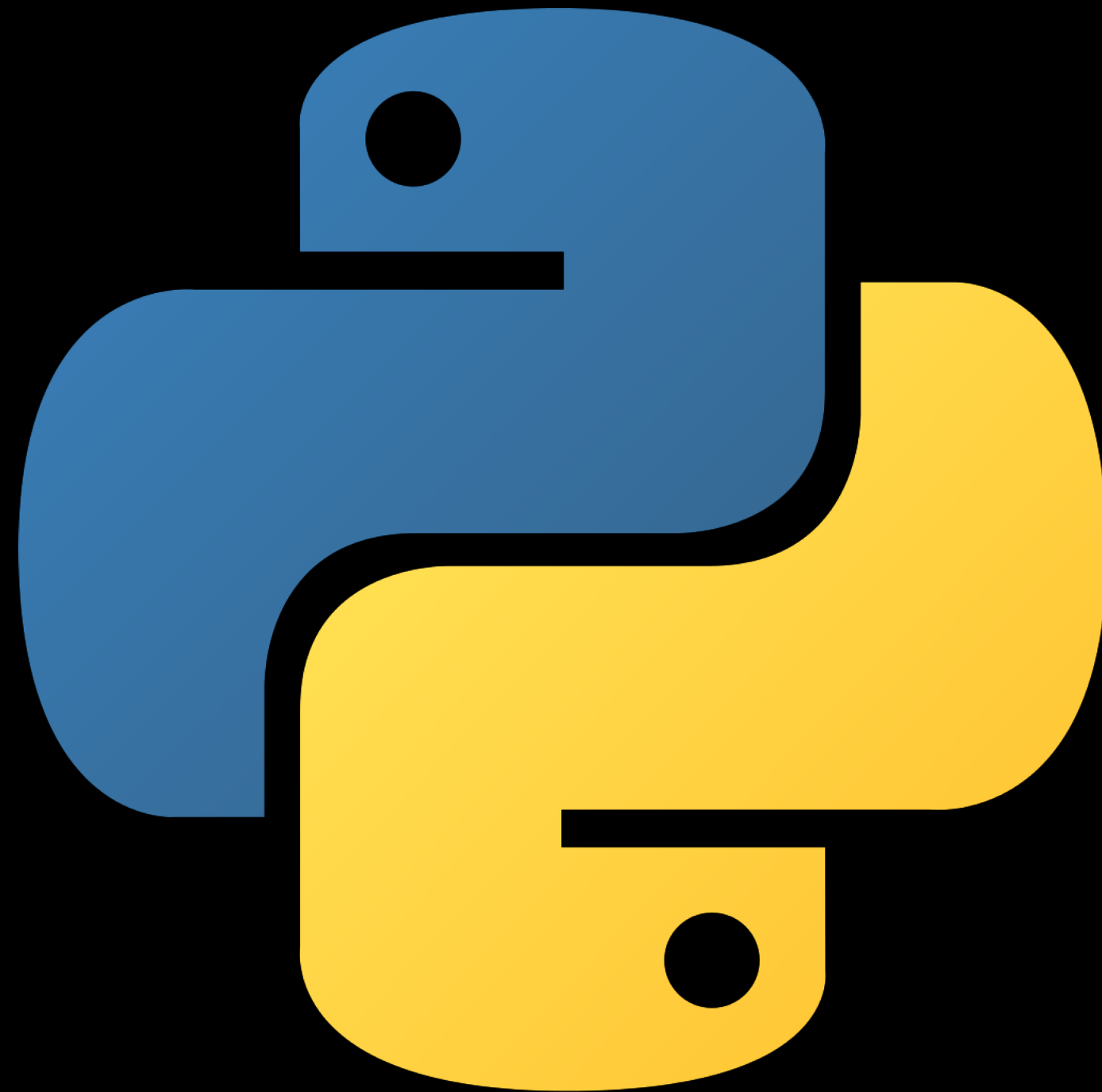
Made only of BCDGIJLMNOPSUWZ

Time Out for Announcements

Setting up Python



Setting up Python



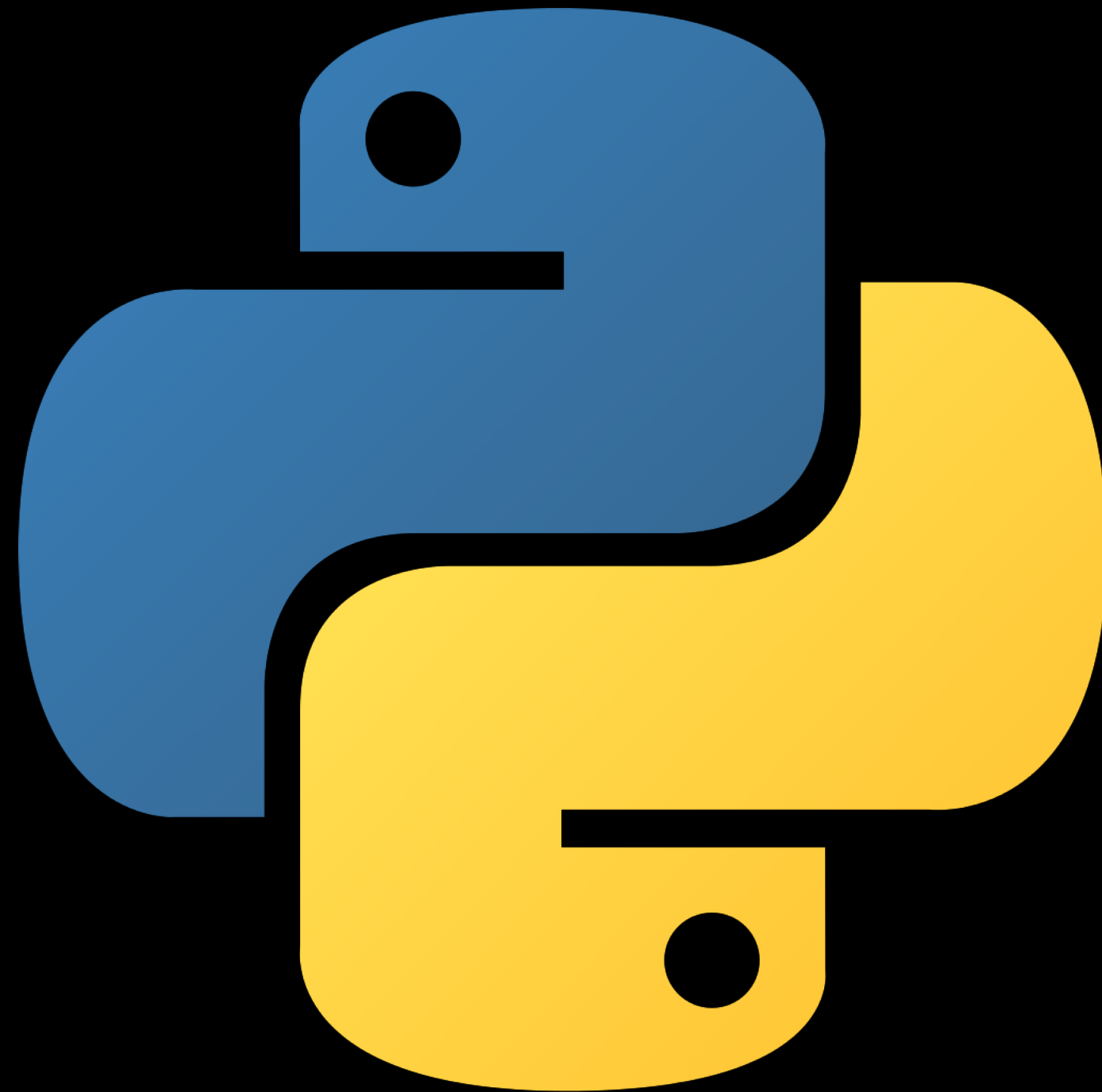
Python 3.7.2

Using Virtual Environments

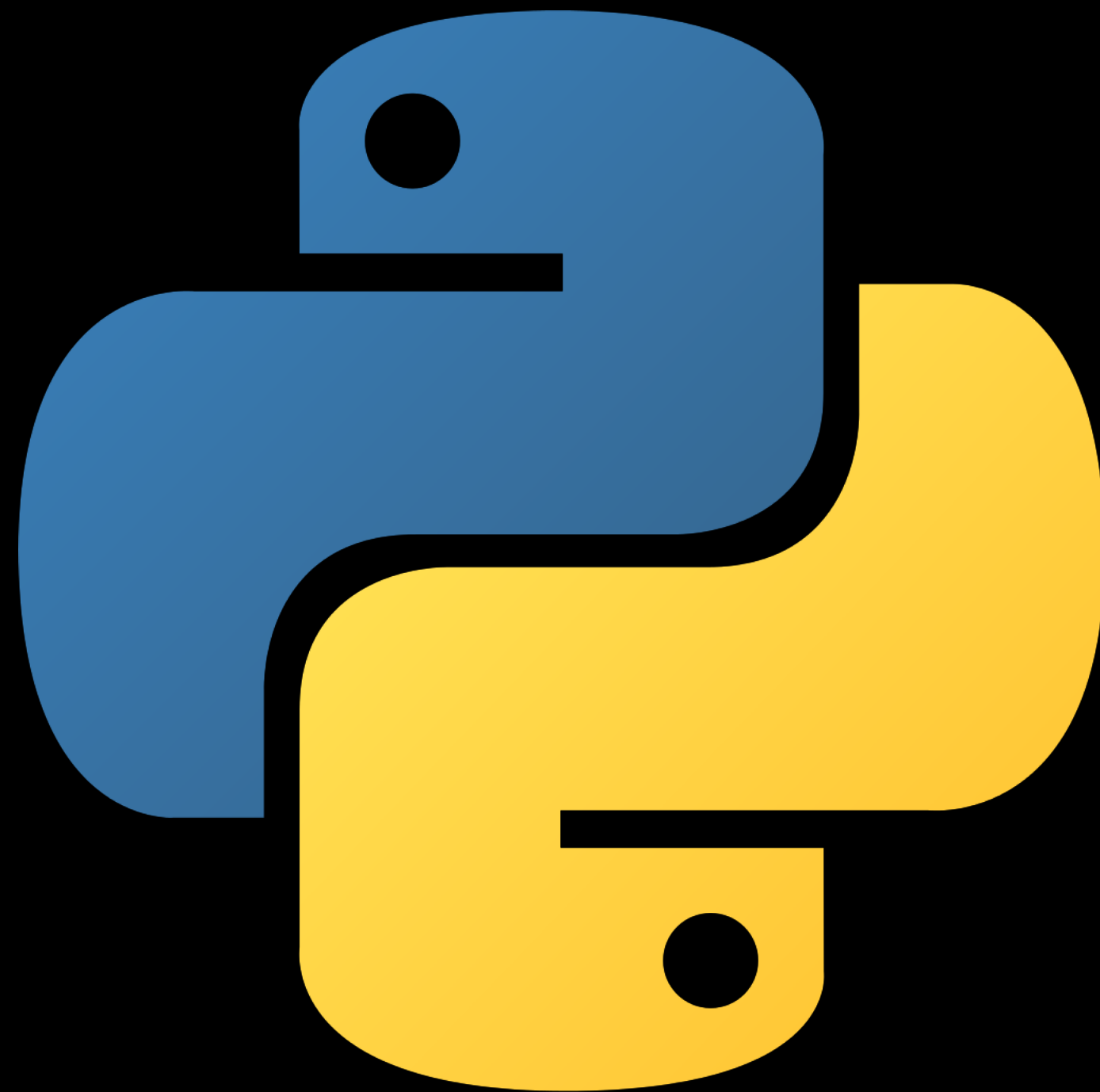
Detailed Instructions

Stuck? See us after, or Piazza.

Assignment 0



Assignment 0



Solidify Python Tools

Due Thursday @ 4:20

Submission via Paperless

Piazza



1 min avg. response time

Piazza



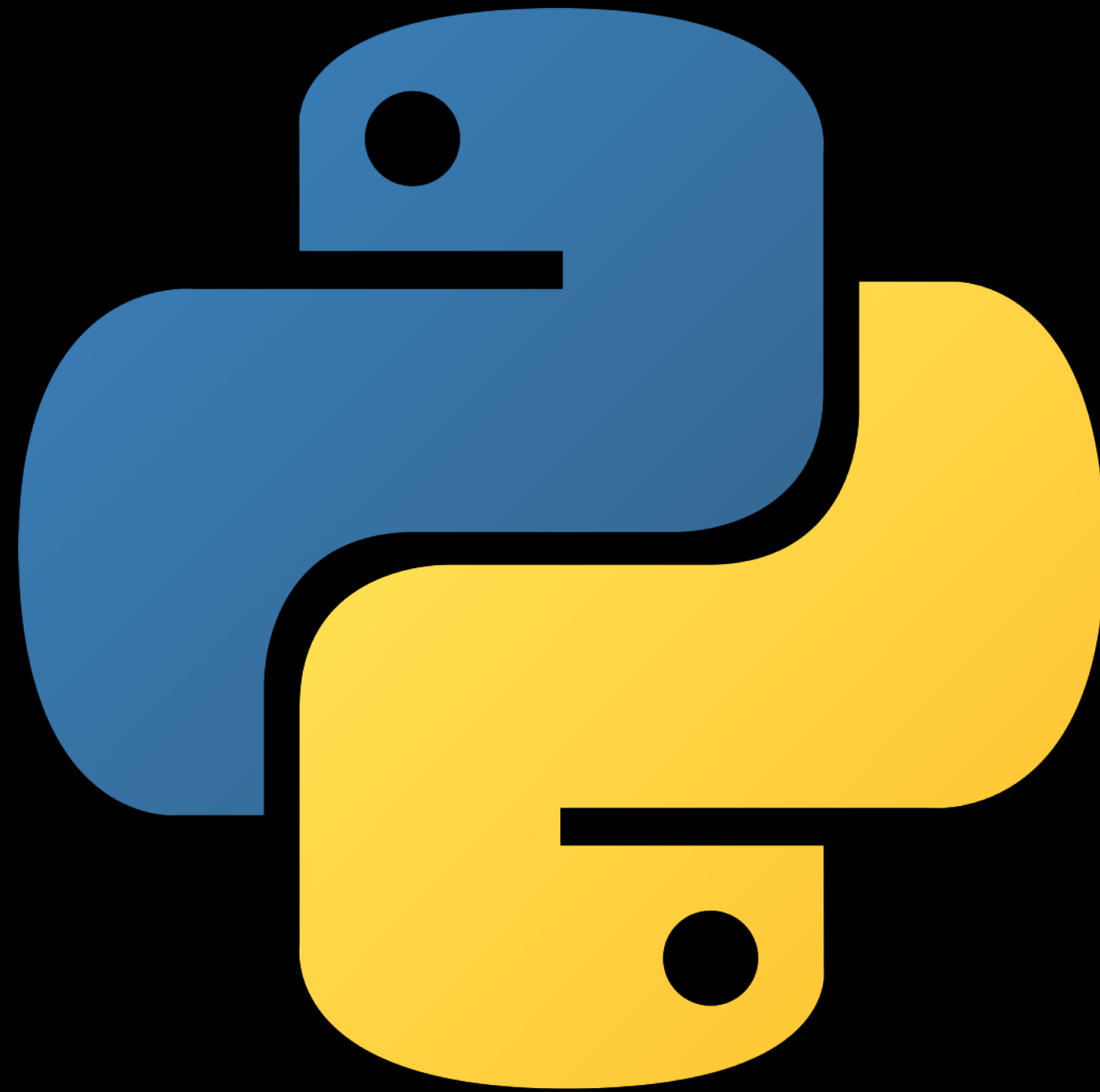
1 min avg. response time

CS 41 on Piazza

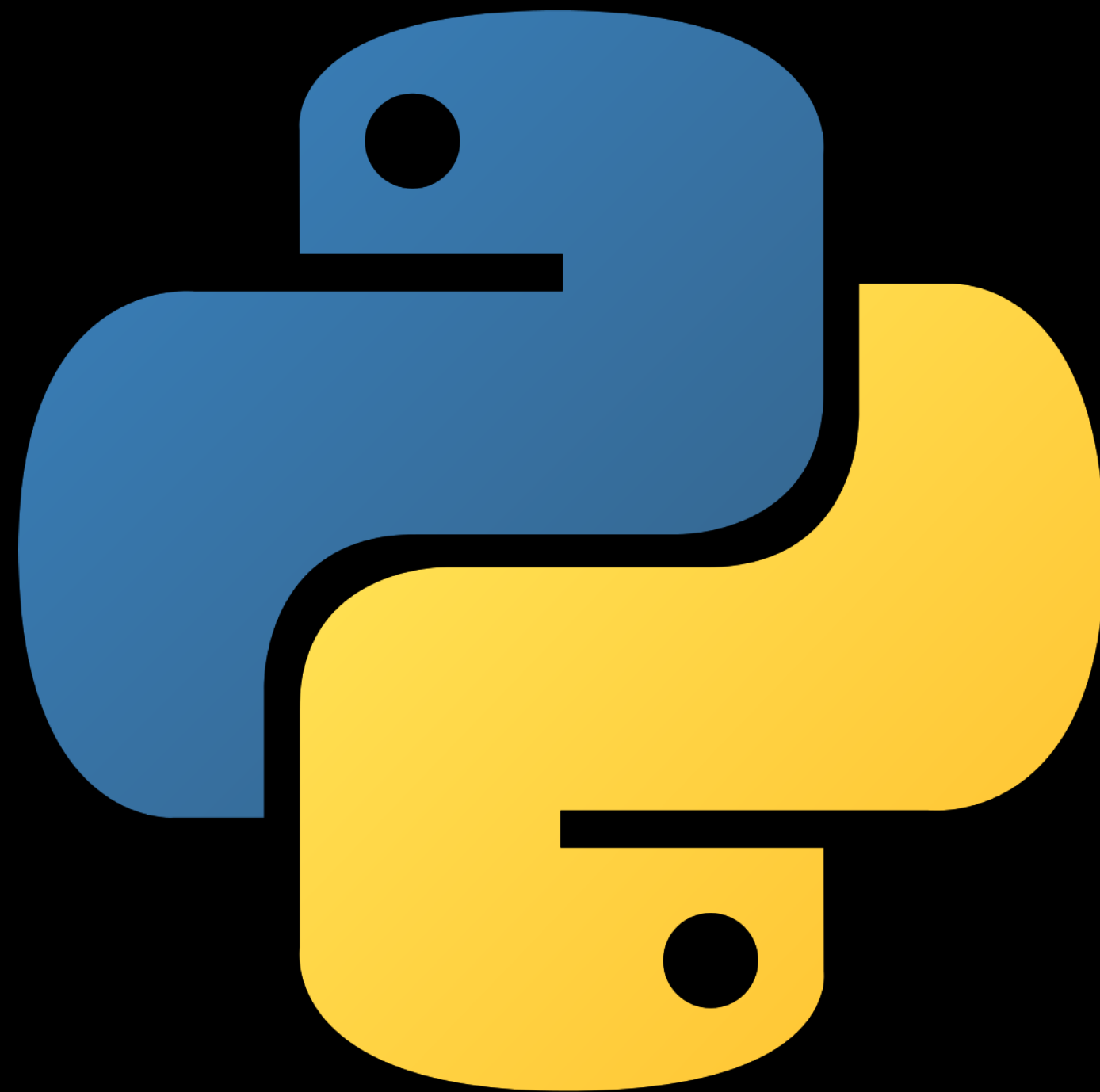
Course Announcements

Ask and Answer Questions

Assignment 1



Assignment 1



Cryptographics Tools!

Caesar, Vigenère, MHKC

Practice Data Structures

Thursday's Lab



Thursday's Lab



Fundamentals and More

"Sections" with course staff

Bring a charged computer!

Enrollment Update

Back to Python!

Back to Python!*

* Follow along with the examples!

Data Structures



Lists

Dictionaries

Tuples

Sets

Advanced Looping

Comprehensions

Lists

Finite, ordered, mutable
sequence of elements

List

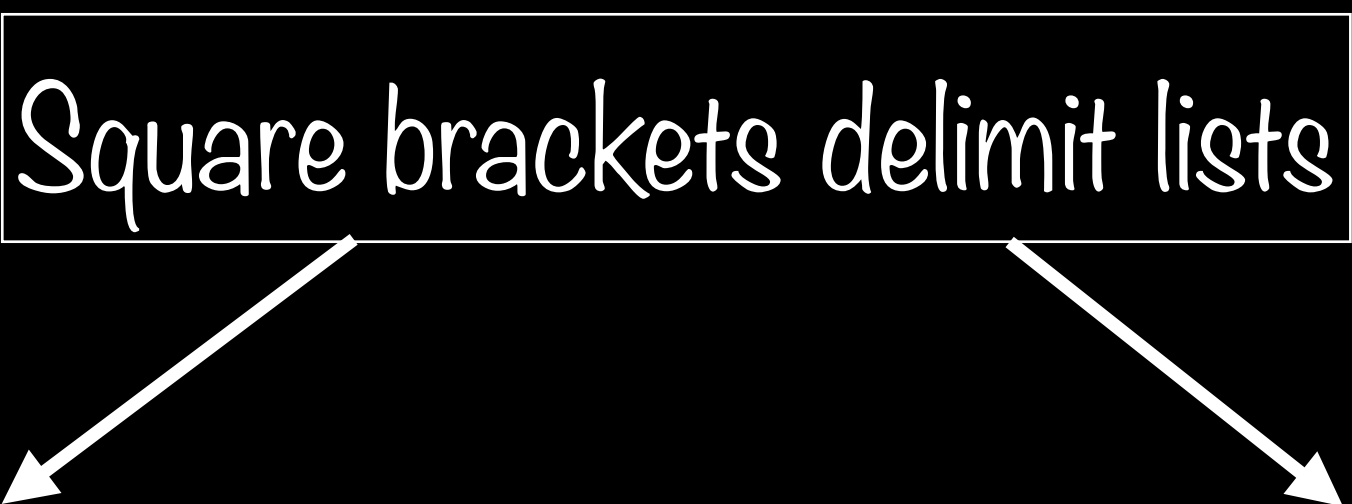
Lists

```
easy_as = [1, 2, 3]
```

Lists

easy_as = [1, 2, 3]

Square brackets delimit lists



The diagram consists of a rectangular box containing the text "Square brackets delimit lists". Two white arrows originate from the bottom-left and bottom-right corners of this box. The left arrow points to the opening square bracket of the list in the code below. The right arrow points to the closing square bracket of the list.

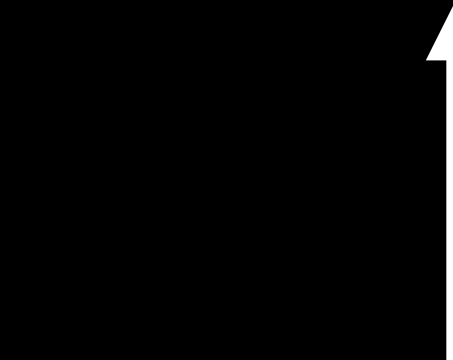
Lists

easy_as

=

[1, 2, 3]

Square brackets delimit lists



Commas separate elements

Basic Lists

Basic Lists

```
# Create a new list
```

```
empty = []
```

```
letters = ['a', 'b', 'c', 'd']
```

```
numbers = [2, 3, 5]
```

Basic Lists

```
# Create a new list
```

```
empty = []
```

```
letters = ['a', 'b', 'c', 'd']
```

```
numbers = [2, 3, 5]
```

```
# Lists can contain elements of different types
```

```
mixed = [4, 5, "seconds"]
```

Basic Lists

```
# Create a new list
```

```
empty = []
```

```
letters = ['a', 'b', 'c', 'd']
```

```
numbers = [2, 3, 5]
```

```
# Lists can contain elements of different types
```

```
mixed = [4, 5, "seconds"]
```

```
# Append elements to the end of a list
```

```
numbers.append(7)    # numbers == [2, 3, 5, 7]
```

```
numbers.append(11)   # numbers == [2, 3, 5, 7, 11]
```

Inspecting List Elements

Inspecting List Elements

Access elements at a particular index

numbers[0] # => 2

numbers[-1] # => 11

Inspecting List Elements

Access elements at a particular index

numbers[0] # => 2

numbers[-1] # => 11

You can also slice lists – the usual rules apply

letters[:3] # => ['a', 'b', 'c']

numbers[1:-1] # => [3, 5, 7]

Nested Lists

Nested Lists

Lists really can contain anything – even other lists!

```
x = [letters, numbers]
```

```
x # => [['a', 'b', 'c', 'd'], [2, 3, 5, 7, 11]]
```

Nested Lists

Lists really can contain anything – even other lists!

```
x = [letters, numbers]
```

```
x # => [['a', 'b', 'c', 'd'], [2, 3, 5, 7, 11]]
```

```
x[0] # => ['a', 'b', 'c', 'd']
```

Nested Lists

Lists really can contain anything – even other lists!

```
x = [letters, numbers]
```

```
x # => [['a', 'b', 'c', 'd'], [2, 3, 5, 7, 11]]
```

```
x[0] # => ['a', 'b', 'c', 'd']
```

```
x[0][1] # => 'b'
```

Nested Lists

Lists really can contain anything – even other lists!

```
x = [letters, numbers]
```

```
x # => [['a', 'b', 'c', 'd'], [2, 3, 5, 7, 11]]
```

```
x[0] # => ['a', 'b', 'c', 'd']
```

```
x[0][1] # => 'b'
```

```
x[1][2:] # => [5, 7, 11]
```

List Method Reference

List Method Reference

```
# Extend list by appending elements from the iterable  
my_list.extend(iterable)
```

List Method Reference

```
# Extend list by appending elements from the iterable  
my_list.extend(iterable)
```

```
# Insert object before index  
my_list.insert(index, object)
```


List Method Reference

Extend list by appending elements from the iterable
`my_list.extend(iterable)`

Insert object before index
`my_list.insert(index, object)`

Remove first occurrence of value, or raise ValueError
`my_list.remove(value)`

List Method Reference

Extend list by appending elements from the iterable
`my_list.extend(iterable)`

Insert object before index
`my_list.insert(index, object)`

Remove first occurrence of value, or raise ValueError
`my_list.remove(value)`

Remove all items
`my_list.clear()`

More List Methods

More List Methods

```
# Return number of occurrences of value  
my_list.count(value)
```

More List Methods

Return number of occurrences of value

```
my_list.count(value)
```

Return first index of value, or raise ValueError

```
my_list.index(value, [start, [stop]])
```

More List Methods

Return number of occurrences of value

```
my_list.count(value)
```

Return first index of value, or raise ValueError

```
my_list.index(value, [start, [stop]])
```

Remove, return item at index (def. last) or IndexError

```
my_list.pop([index])
```

More List Methods

Return number of occurrences of value

```
my_list.count(value)
```

Return first index of value, or raise ValueError

```
my_list.index(value, [start, [stop]])
```

Remove, return item at index (def. last) or IndexError

```
my_list.pop([index])
```

Stable sort **in place**

```
my_list.sort(key=None, reverse=False)
```

More List Methods

Return number of occurrences of value

```
my_list.count(value)
```

Return first index of value, or raise ValueError

```
my_list.index(value, [start, [stop]])
```

Remove, return item at index (def. last) or IndexError

```
my_list.pop([index])
```

Stable sort *in place*

```
my_list.sort(key=None, reverse=False)
```

Reverse *in place*.

```
my_list.reverse()
```


General Queries on Iterables

General Queries on Iterables

Length (len)

len([]) # => 0

len("python") # => 6

len([4, 5, "seconds"]) # => 3

General Queries on Iterables

Length (len)

len([]) # => 0

len("python") # => 6

len([4, 5, "seconds"]) # => 3

Membership (in)

0 in [] # => False

'y' in 'python' # => True

'minutes' in [4, 5, 'seconds'] # => False

Dictionaries

Mutable map from hashable
values to arbitrary objects

Dictionary

Keys can be a variety of types,
as long as they are hashable

Mutable map from hashable
values to arbitrary objects

Dictionary

Keys can be a variety of types,
as long as they are hashable

Mutable map from hashable
values to arbitrary objects

Values can be a variety of types too

Dictionary

Create a Dictionary

Create a Dictionary

```
empty = {}
```

```
type(empty)      # => dict
```

```
empty == dict()  # => True
```

Create a Dictionary

```
empty = {}
```

```
type(empty)      # => dict
```

```
empty == dict()  # => True
```

```
a = dict(one=1, two=2, three=3)
```

```
b = {"one": 1, "two": 2, "three": 3}
```

```
a == b  # => True
```

Access and Mutate

Access and Mutate

```
d = {"one": 1, "two": 2, "three": 3}
```

Access and Mutate

```
d = {"one": 1, "two": 2, "three": 3}
```

```
# Get
```

```
d['one'] # => 1
```

```
d['five'] # raises KeyError
```

Access and Mutate

```
d = {"one": 1, "two": 2, "three": 3}
```

```
# Get
```

```
d['one'] # => 1
```

```
d['five'] # raises KeyError
```

```
# Set
```

```
d['two'] = 22 # Modify an existing key
```

```
d['four'] = 4 # Add a new key
```

Get with Default

Get with Default

```
d = {"CS": [106, 107, 110], "MATH": [51, 113]}
```


Get with Default

```
d = {"CS": [106, 107, 110], "MATH": [51, 113]}  
d["COMPSCI"] # raises KeyError
```

Get with Default

```
d = {"CS": [106, 107, 110], "MATH": [51, 113]}
```

```
d["COMPSCI"] # raises KeyError
```

Use `get()` method to avoid the `KeyError`

```
d.get("CS") # => [106, 107, 110]
```

```
d.get("PHIL") # => None (not a KeyError!)
```

Get with Default

```
d = {"CS": [106, 107, 110], "MATH": [51, 113]}
```

```
d["COMPSCI"] # raises KeyError
```

Use `get()` method to avoid the `KeyError`

```
d.get("CS") # => [106, 107, 110]
```

```
d.get("PHIL") # => None (not a KeyError!)
```

```
english_classes = d.get("ENGLISH", [])
```

```
num_english = len(english_classes)
```

Works even if there were no English classes in our dictionary!

Delete

Delete

```
d = {"one": 1, "two": 2, "three": 3}
```

Delete

```
d = {"one": 1, "two": 2, "three": 3}
```

```
del d["one"]
```

Raises `KeyError` if invalid key

Delete

```
d = {"one": 1, "two": 2, "three": 3}
```

```
del d["one"]
```

Raises `KeyError` if invalid key

```
d.pop("three", default) # => 3
```

Remove and return `d['three']`
or default value if not in the map

Delete

```
d = {"one": 1, "two": 2, "three": 3}
```

```
del d["one"]
```

Raises `KeyError` if invalid key

```
d.pop("three", default) # => 3
```

Remove and return `d['three']`
or default value if not in the map

```
d.popitem() # => ("two", 2)
```

Remove and return an arbitrary
(key, value) pair.
Useful for destructive iteration

Dictionaries

Dictionaries

```
d = {"one": 1, "two": 2, "three": 3}
```

Dictionaries

```
d = {"one": 1, "two": 2, "three": 3}
```

```
d.keys()
```

```
d.values()
```

```
d.items()
```

Dictionaries

```
d = {"one": 1, "two": 2, "three": 3}
```

```
d.keys()
```

```
d.values()
```

```
d.items()
```

These dictionary views are dynamic,
reflecting changes in the underlying dictionary!

Dictionaries

```
d = {"one": 1, "two": 2, "three": 3}
```

```
d.keys()
```

```
d.values()
```

```
d.items()
```

```
len(d.keys()) # => 3
```

These dictionary views are dynamic,
reflecting changes in the underlying dictionary!

Dictionaries

```
d = {"one": 1, "two": 2, "three": 3}
```

```
d.keys()
```

```
d.values()
```

```
d.items()
```

```
len(d.keys()) # => 3
```

```
('one', 1) in d.items()
```

These dictionary views are dynamic,
reflecting changes in the underlying dictionary!

Dictionaries

```
d = {"one": 1, "two": 2, "three": 3}
```

```
d.keys()
```

```
d.values()
```

```
d.items()
```

```
len(d.keys()) # => 3
```

```
('one', 1) in d.items()
```

```
for value in d.values():  
    print(value)
```

These dictionary views are dynamic,
reflecting changes in the underlying dictionary!

Dictionaries

```
d = {"one": 1, "two": 2, "three": 3}
```

```
d.keys()
```

```
d.values()
```

```
d.items()
```

These dictionary views are dynamic,
reflecting changes in the underlying dictionary!

```
len(d.keys()) # => 3
```

```
('one', 1) in d.items()
```

```
for value in d.values():
```

```
    print(value)
```

```
keys_list = list(d.keys())
```


Dictionaries

```
d = {"one": 1, "two": 2, "three": 3}
```

```
d.keys()
```

```
d.values()
```

```
d.items()
```

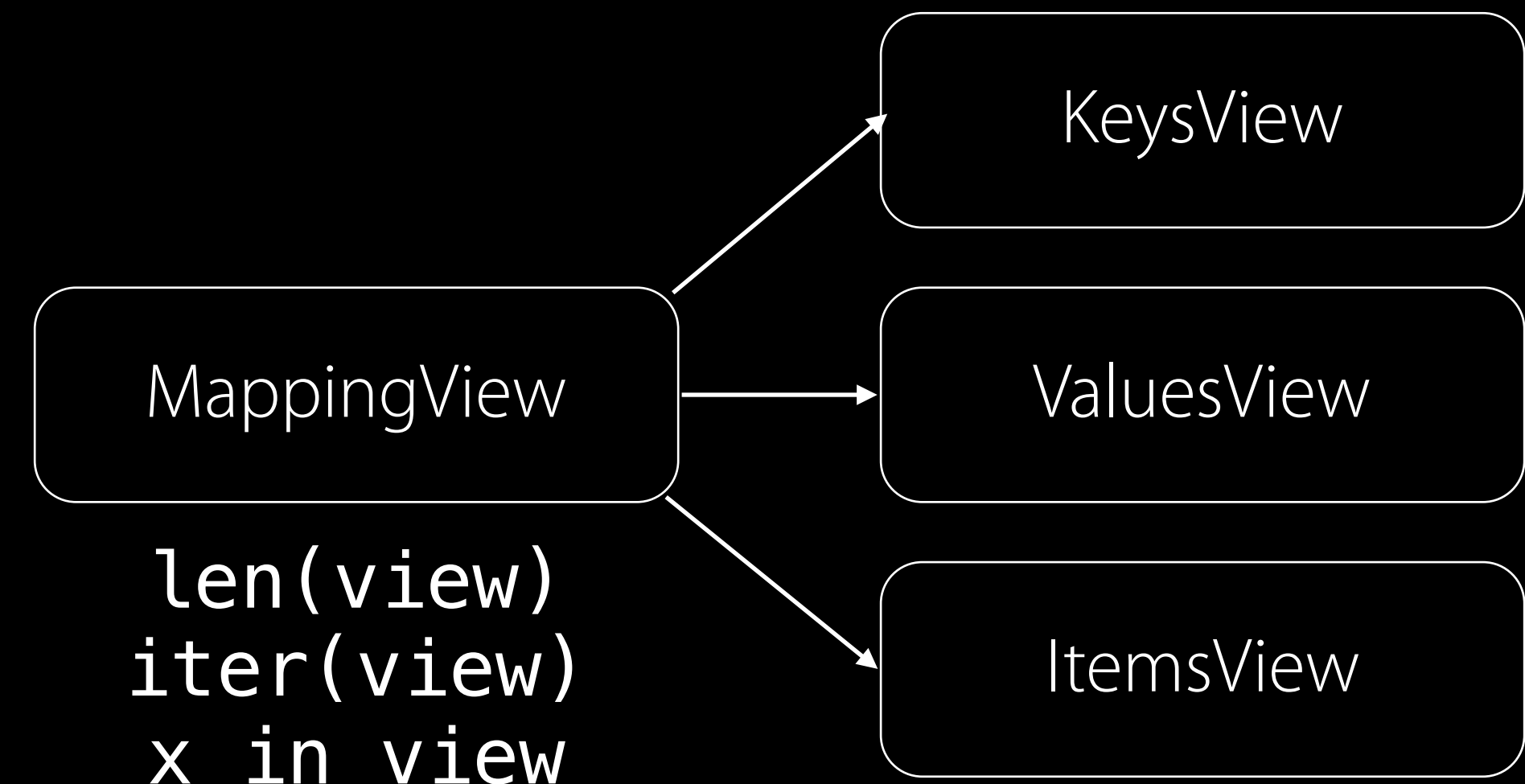
```
len(d.keys()) # => 3
```

```
('one', 1) in d.items()
```

```
for value in d.values():  
    print(value)
```

```
keys_list = list(d.keys())
```

These dictionary views are dynamic,
reflecting changes in the underlying dictionary!



Common Dict Operations

Common Dict Operations

`len(d)`

Common Dict Operations

`len(d)`

`key in d` # equiv. to ``key in d.keys()``

Common Dict Operations

`len(d)`

`key in d` # equiv. to ``key in d.keys()``

`value in d.values()`

Common Dict Operations

`len(d)`

`key in d` # equiv. to ``key in d.keys()``

`value in d.values()`

`d.copy()`

Common Dict Operations

`len(d)`

`key in d` # equiv. to ``key in d.keys()``

`value in d.values()`

`d.copy()`

`d.clear()`

Common Dict Operations

```
len(d)
```

```
key in d # equiv. to `key in d.keys()`
```

```
value in d.values()
```

```
d.copy()
```

```
d.clear()
```

```
for key in d: # equiv. to `for key in d.keys():`  
    print(key)
```


Tuples

Immutable Sequences

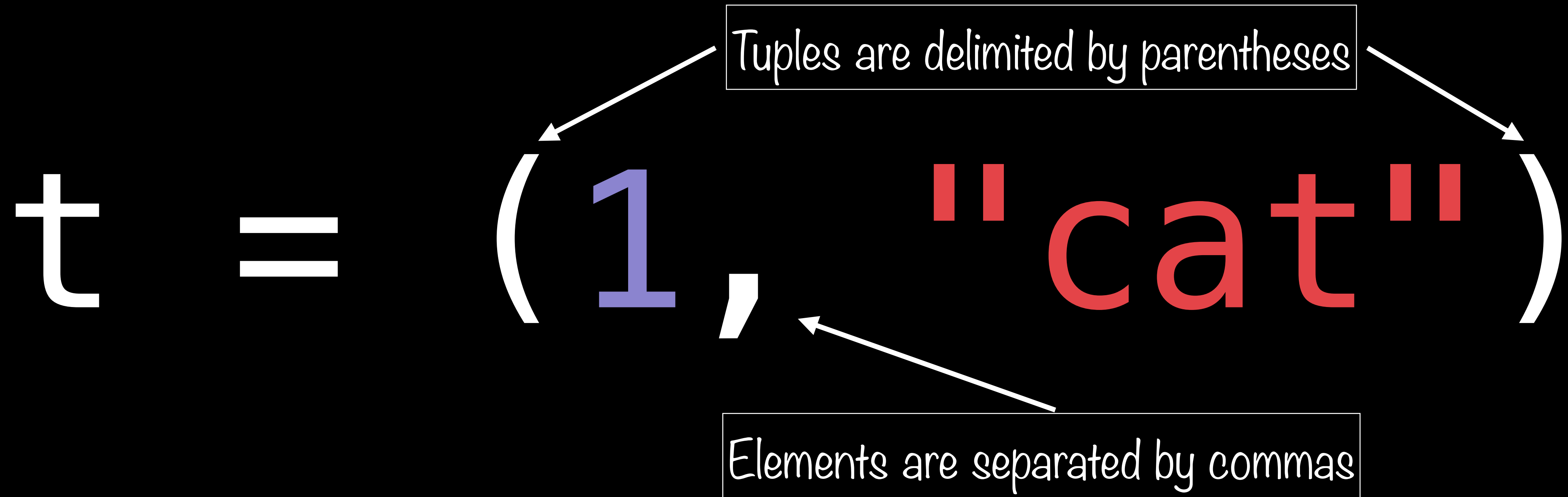
Tuple

t = (1, "cat")

`t = (1, "cat")`

Tuples are delimited by parentheses

The diagram illustrates a tuple in Python. It shows the variable `t` assigned to a tuple containing the integer `1` and the string `"cat"`. The tuple is enclosed in parentheses. A text box with the annotation "Tuples are delimited by parentheses" has two arrows pointing to the opening and closing parentheses of the tuple.



Primary Motivations

Primary Motivations

Store collections of heterogeneous data

Think `struct`- or `SQL`-like objects

Primary Motivations

Store collections of heterogeneous data

Think `struct`- or `SQL`-like objects

"Freeze" sequence to ensure hashability

Tuples can be dictionary keys, but lists cannot

Primary Motivations

Store collections of heterogeneous data

Think `struct`- or `SQL`-like objects

"Freeze" sequence to ensure hashability

Tuples can be dictionary keys, but lists cannot

Enforce immutability for fixed-size collections

Tuples

Tuples

```
fish = (1, 2, "red", "blue")
```

Tuples

```
fish = (1, 2, "red", "blue")
```

```
fish[0]          # => 1
```

Tuples

```
fish = (1, 2, "red", "blue")  
fish[0]          # => 1  
fish[0] = 7      # Raises a TypeError
```

You can't change any
elements in a tuple!

Tuples

```
fish = (1, 2, "red", "blue")  
fish[0]          # => 1  
fish[0] = 7      # Raises a TypeError
```

You can't change any
elements in a tuple!

```
len(fish)        # => 4  
fish[:2]         # => (1, 2)  
"red" in fish    # => True
```

Although the usual sequence
methods still work

Argument Packing and Unpacking

Argument Packing and Unpacking

```
t = 12345, 54321, 'hello!'
```


Argument Packing and Unpacking

```
t = 12345, 54321, 'hello!'
```

Comma-separated Rvalues
are converted to a tuple

Argument Packing and Unpacking

```
t = 12345, 54321, 'hello!'
print(t) # (12345, 54321, 'hello!')
type(t)  # => tuple
```

Comma-separated Rvalues
are converted to a tuple

Argument Packing and Unpacking

```
t = 12345, 54321, 'hello!'
print(t) # (12345, 54321, 'hello!')
type(t)  # => tuple
```

Comma-separated Rvalues
are converted to a tuple

```
x, y, z = t
```

Argument Packing and Unpacking

```
t = 12345, 54321, 'hello!'
print(t) # (12345, 54321, 'hello!')
type(t)  # => tuple
```

Comma-separated Rvalues
are converted to a tuple

```
x, y, z = t
```

Comma-separated Lvalues
are unpacked automatically

Argument Packing and Unpacking

```
t = 12345, 54321, 'hello!'
print(t) # (12345, 54321, 'hello!')
type(t)  # => tuple
```

Comma-separated Rvalues
are converted to a tuple

```
x, y, z = t
x # => 12345
y # => 54321
z # => 'hello!'
```

Comma-separated Lvalues
are unpacked automatically

Swapping Values

Have $x = 5$
 $y = 6$

Want $x = 6$
 $y = 5$

Swapping Values

Have $x = 5$
 $y = 6$

Want $x = 6$
 $y = 5$

```
temp = x
x = y
y = temp

print(x, y)
# => 6 5
```

Temporary
Variable

Swapping Values

Have $x = 5$
 $y = 6$

Want $x = 6$
 $y = 5$

```
temp = x  
x = y  
y = temp
```

```
print(x, y)  
# => 6 5
```

Temporary
Variable

```
x = x ^ y  
y = x ^ y  
x = x ^ y
```

```
print(x, y)  
# => 6 5
```

XOR
Magic

Swapping Values

Have $x = 5$
 $y = 6$

Want $x = 6$
 $y = 5$

```
temp = x
x = y
y = temp
```

```
print(x, y)
# => 6 5
```

Temporary
Variable

```
x = x ^ y
y = x ^ y
x = x ^ y
```

```
print(x, y)
# => 6 5
```

XOR
Magic

```
x, y = y, x
```

```
print(x, y)
# => 6 5
```

Tuple Packing

Have $x = 5$
 $y = 6$

Want $x = 6$
 $y = 5$

$x, y = y, x$

Have $x = 5$
 $y = 6$

Want $x = 6$
 $y = 5$

First, y , x is packed into the tuple $(6, 5)$

$x, y = y, x$

Have $x = 5$ Want $x = 6$
 $y = 6$ $y = 5$

First, y, x is packed into the tuple $(6, 5)$

$x, y = y, x$

Then, $(6, 5)$ is unpacked into the variables x and y respectively

Fibonacci Sequence

Fibonacci Sequence

```
def fib(n):  
    """Prints the first n Fibonacci numbers."""
```

Fibonacci Sequence

```
def fib(n):  
    """Prints the first n Fibonacci numbers."""  
    a, b = 0, 1
```

Fibonacci Sequence

```
def fib(n):  
    """Prints the first n Fibonacci numbers."""  
    a, b = 0, 1  
    for i in range(n):  
        print(i, a)
```


Fibonacci Sequence

```
def fib(n):  
    """Prints the first n Fibonacci numbers."""  
    a, b = 0, 1  
    for i in range(n):  
        print(i, a)  
        a, b = b, a + b
```

A New Example

A New Example

```
for index, color in enumerate(['red', 'green', 'blue']):  
    print(index, color)
```

A New Example

```
for index, color in enumerate(['red', 'green', 'blue']):  
    print(index, color)
```

```
# =>
```

```
# 0 red
```

```
# 1 green
```

```
# 2 blue
```

A New Example

```
for index, color in enumerate(['red', 'green', 'blue']):  
    print(index, color)
```

```
# =>
```

```
# 0 red
```

```
# 1 green
```

```
# 2 blue
```

This also means you should almost never use
`for i in range(len(sequence)):`

Quirks

Quirks

```
empty = ()  
singleton = ("value",)  
plain_string = "value" # Note plain_string != singleton
```

Quirks

```
empty = ()  
singleton = ("value",)  
plain_string = "value" # Note plain_string != singleton  
len(empty) # => 0  
len(singleton) # => 1
```


Quirks

```
empty = ()  
singleton = ("value",)  
plain_string = "value" # Note plain_string != singleton  
len(empty)          # => 0  
len(singleton)      # => 1  
  
v = ([1, 2, 3], ['a', 'b', 'c'])
```

Quirks

```
empty = ()  
singleton = ("value",)  
plain_string = "value" # Note plain_string != singleton  
len(empty) # => 0  
len(singleton) # => 1  
  
v = ([1, 2, 3], ['a', 'b', 'c'])  
v[0].append(4)
```

Quirks

```
empty = ()  
singleton = ("value",)  
plain_string = "value" # Note plain_string != singleton  
len(empty) # => 0  
len(singleton) # => 1
```

```
v = ([1, 2, 3], ['a', 'b', 'c'])  
v[0].append(4)  
v # => ([1, 2, 3, 4], ['a', 'b', 'c'])
```

Quirks

```
empty = ()  
singleton = ("value",)  
plain_string = "value" # Note plain_string != singleton  
len(empty) # => 0  
len(singleton) # => 1
```

```
v = ([1, 2, 3], ['a', 'b', 'c'])
```

```
v[0].append(4)
```

```
v # => ([1, 2, 3, 4], ['a', 'b', 'c'])
```

Tuples contain (immutable) references
to underlying objects!

Sets

Unordered collection of
distinct hashable elements

Set

Primary Motivations

Primary Motivations

Fast membership testing

$O(1)$ vs. $O(n)$

Primary Motivations

Fast membership testing

$O(1)$ vs. $O(n)$

Eliminate duplicate entries

Primary Motivations

Fast membership testing

$O(1)$ vs. $O(n)$

Eliminate duplicate entries

Easy set operations (intersection, union, etc.)

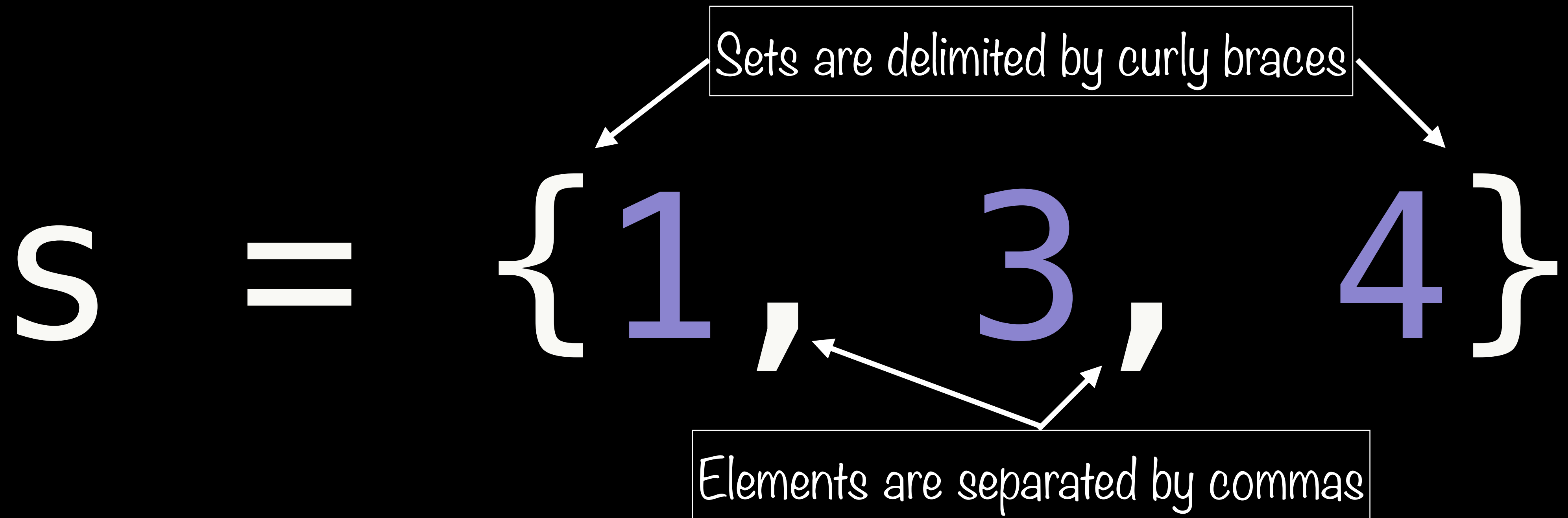
$$S = \{1, 3, 4\}$$

Unordered collection of distinct hashable elements

Sets are delimited by curly braces

$$S = \{1, 3, 4\}$$

Unordered collection of distinct hashable elements



Common Set Operations

Why not {}?

Common Set Operations

```
empty_set = set()
```

Why not {}?

Common Set Operations

```
empty_set = set()
```

```
set_from_list = set([1, 2, 1, 4, 3]) # => {1, 3, 4, 2}
```


Why not {}?

Common Set Operations

```
empty_set = set()
```

```
set_from_list = set([1, 2, 1, 4, 3]) # => {1, 3, 4, 2}
```

```
basket = {"apple", "orange", "apple", "pear", "banana"}
```

Why not {}?

Common Set Operations

```
empty_set = set()
```

```
set_from_list = set([1, 2, 1, 4, 3]) # => {1, 3, 4, 2}
```

```
basket = {"apple", "orange", "apple", "pear", "banana"}
```

```
len(basket)           # => 4
```

Why not {}?

Common Set Operations

```
empty_set = set()
```

```
set_from_list = set([1, 2, 1, 4, 3]) # => {1, 3, 4, 2}
```

```
basket = {"apple", "orange", "apple", "pear", "banana"}
```

```
len(basket) # => 4
```

```
"orange" in basket # => True
```

```
"crabgrass" in basket # => False
```

$O(1)$ membership testing

Why not {}?

Common Set Operations

```
empty_set = set()
```

```
set_from_list = set([1, 2, 1, 4, 3]) # => {1, 3, 4, 2}
```

```
basket = {"apple", "orange", "apple", "pear", "banana"}
```

```
len(basket) # => 4
```

```
"orange" in basket # => True
```

```
"crabgrass" in basket # => False
```

$O(1)$ membership testing

```
for fruit in basket:
```

```
    print(fruit, end='/')
```

```
# => pear/banana/apple/orange/
```

Common Set Operations

Common Set Operations

```
a = set("mississippi")  # {'i', 'm', 'p', 's'}
```

Common Set Operations

```
a = set("mississippi")  # {'i', 'm', 'p', 's'}
```

```
a.add('r')
```

```
a.remove('m')  # raises KeyError if 'm' is not present
```

```
a.discard('x')  # same as remove, except no error
```

Common Set Operations

```
a = set("mississippi") # {'i', 'm', 'p', 's'}
```

```
a.add('r')
```

```
a.remove('m') # raises KeyError if 'm' is not present
```

```
a.discard('x') # same as remove, except no error
```

```
a.pop() # => 's' (or 'i' or 'p')
```

```
a.clear()
```


Common Set Operations

```
a = set("mississippi") # {'i', 'm', 'p', 's'}
```

```
a.add('r')
```

```
a.remove('m') # raises KeyError if 'm' is not present
```

```
a.discard('x') # same as remove, except no error
```

```
a.pop() # => 's' (or 'i' or 'p')
```

```
a.clear()
```

```
len(a) # => 0
```

Common Set Operations

Common Set Operations

```
a = set("abracadabra") # {'a', 'r', 'b', 'c', 'd'}  
b = set("alacazam")    # {'a', 'm', 'c', 'l', 'z'}
```

Common Set Operations

```
a = set("abracadabra") # {'a', 'r', 'b', 'c', 'd'}  
b = set("alacazam")   # {'a', 'm', 'c', 'l', 'z'}
```

```
# Set difference
```

```
a - b # => {'r', 'd', 'b'}
```

Common Set Operations

```
a = set("abracadabra") # {'a', 'r', 'b', 'c', 'd'}  
b = set("alacazam")    # {'a', 'm', 'c', 'l', 'z'}
```

Set difference

```
a - b # => {'r', 'd', 'b'}
```

Union

```
a | b # => {'a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'}
```

Common Set Operations

```
a = set("abracadabra") # {'a', 'r', 'b', 'c', 'd'}  
b = set("alacazam")   # {'a', 'm', 'c', 'l', 'z'}
```

Set difference

```
a - b # => {'r', 'd', 'b'}
```

Union

```
a | b # => {'a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'}
```

Intersection

```
a & b # => {'a', 'c'}
```

Common Set Operations

```
a = set("abracadabra") # {'a', 'r', 'b', 'c', 'd'}  
b = set("alacazam")   # {'a', 'm', 'c', 'l', 'z'}
```

Set difference

```
a - b # => {'r', 'd', 'b'}
```

Union

```
a | b # => {'a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'}
```

Intersection

```
a & b # => {'a', 'c'}
```

Symmetric Difference

```
a ^ b # => {'r', 'd', 'b', 'm', 'z', 'l'}
```

Rewriting `is_efficient`

Rewriting is_efficient

```
EFFICIENT_LETTERS = "BCDGIJLMNOPSUWZ"
```

Rewriting is_efficient

```
EFFICIENT_LETTERS = "BCDGIJLMNOPSUWZ"
```

```
def is_efficient(word):
```

Rewriting is_efficient

```
EFFICIENT_LETTERS = "BCDGIJLMNOPSUWZ"
```

```
def is_efficient(word):  
    for letter in word:
```

Rewriting is_efficient

```
EFFICIENT_LETTERS = "BCDGIJLMNOPSUWZ"
```

```
def is_efficient(word):  
    for letter in word:  
        if letter not in EFFICIENT_LETTERS:
```

Rewriting is_efficient

```
EFFICIENT_LETTERS = "BCDGIJLMNOPSUWZ"
```

```
def is_efficient(word):  
    for letter in word:  
        if letter not in EFFICIENT_LETTERS:  
            return False
```

Rewriting is_efficient

```
EFFICIENT_LETTERS = "BCDGIJLMNOPSUWZ"
```

```
def is_efficient(word):  
    for letter in word:  
        if letter not in EFFICIENT_LETTERS:  
            return False  
    return True
```

Rewriting is_efficient

Rewriting is_efficient

```
EFFICIENT_LETTERS = set("BCDGIJLMNOPSUUVWZ")
```


Rewriting is_efficient

```
EFFICIENT_LETTERS = set("BCDGIJLMNOPSUUVWZ")
```

```
def is_efficient(word):  
    return set(word) <= EFFICIENT_LETTERS
```

Rewriting is_efficient

```
EFFICIENT_LETTERS = set("BCDGIJLMNOPSUWZ")
```

```
def is_efficient(word):  
    return set(word) <= EFFICIENT_LETTERS
```

Is the set of letters in this word a subset of the efficient letters?

Looping Techniques

Items in Dictionary

Items in Dictionary

```
knights = {'gallahad': 'the pure', 'robin': 'the brave'}
```

Items in Dictionary

```
knight = {'gallahad': 'the pure', 'robin': 'the brave'}  
for k, v in knight.items():  
    print(k, v)
```

Items in Dictionary

```
knights = {'gallahad': 'the pure', 'robin': 'the brave'}  
for k, v in knights.items():  
    print(k, v)
```

```
# =>
```

```
# gallahad the pure
```

```
# robin the brave
```

zip

zip

```
questions = ['name', 'quest', 'favorite color']
```

```
answers = ['Lancelot', 'To seek the holy grail', 'Blue']
```

zip

```
questions = ['name', 'quest', 'favorite color']  
answers = ['Lancelot', 'To seek the holy grail', 'Blue']  
for q, a in zip(questions, answers):  
    print('What is your {0}? {1}.'.format(q, a))
```

zip

```
questions = ['name', 'quest', 'favorite color']  
answers = ['Lancelot', 'To seek the holy grail', 'Blue']  
for q, a in zip(questions, answers):  
    print('What is your {0}? {1}.'.format(q, a))
```

The `zip()` function generates pairs of entries from its arguments.

zip

```
questions = ['name', 'quest', 'favorite color']  
answers = ['Lancelot', 'To seek the holy grail', 'Blue']  
for q, a in zip(questions, answers):  
    print('What is your {0}? {1}.'.format(q, a))
```

The `zip()` function generates pairs of entries from its arguments.

```
# =>
```

```
# What is your name? Lancelot.
```

```
# What is your quest? To seek the holy grail.
```

```
# What is your favorite color? Blue.
```

Reverse Iteration

Reverse Iteration

```
for i in reversed(range(1, 10, 2)):  
    print(i, end=', ')
```

Reverse Iteration

```
for i in reversed(range(1, 10, 2)):  
    print(i, end=', ')
```

=>

9, 7, 5, 3, 1,

Reverse Iteration

```
for i in reversed(range(1, 10, 2)):  
    print(i, end=', ')
```

```
# =>
```

```
# 9, 7, 5, 3, 1,
```

To loop over a sequence in reverse,
first specify the sequence in a forward direction
and then call the `reversed()` function.

Sorted Iteration

Sorted Iteration

```
basket = ['pear', 'banana', 'orange', 'pear', 'apple']
```

Sorted Iteration

```
basket = ['pear', 'banana', 'orange', 'pear', 'apple']  
for fruit in sorted(basket):  
    print(fruit)
```

Sorted Iteration

```
basket = ['pear', 'banana', 'orange', 'pear', 'apple']
```

```
for fruit in sorted(basket):  
    print(fruit)
```

```
# =>
```

```
# apple
```

```
# banana
```

```
# orange
```

```
# pear
```

```
# pear
```

Sorted Iteration

```
basket = ['pear', 'banana', 'orange', 'pear', 'apple']
```

```
for fruit in sorted(basket):  
    print(fruit)
```

```
# =>
```

```
# apple
```

```
# banana
```

```
# orange
```

```
# pear
```

```
# pear
```

To loop over a sequence in sorted order,
use the `sorted()` function which returns
a new sorted list while leaving the source unaltered.

Comprehensions

Concise syntax for creating data structures

Comprehensions

Example

Example

```
squares = []
```

Example

```
squares = []  
for x in range(100):
```

Example

```
squares = []  
for x in range(100):  
    squares.append(x**2)
```

Example

```
squares = []
```

```
for x in range(100):
```

```
    squares.append(x**2)
```

```
print(squares[:5] + squares[-5:])
```

```
# [0, 1, 4, 9, 16, 9025, 9216, 9409, 9604, 9801]
```

[

]

[x ** 2

]

```
[x ** 2  
for x in ]
```

```
[x ** 2  
for x in  
range(100)]
```



```
[f(xs) for xs in iter]
```

Square brackets indicate that we're building a list

```
[f(xs) for xs in iter]
```

Square brackets indicate that we're building a list

Loop over the specified iterable

```
[f(xs) for xs in iter]
```

Square brackets indicate that we're building a list

Loop over the specified iterable

[f(xs) for xs in iter]

Apply some operation to
the loop variable(s) to generate
new list elements

```
[f(xs) for xs in iter if pred(xs)]
```

Only keep elements that
satisfy a predicate condition

Examples of List Comprehensions

Examples of List Comprehensions

```
[word.lower() for word in sentence]
```

Examples of List Comprehensions

```
[word.lower() for word in sentence]
```

```
[word for word in sentence if len(word) > 8]
```


Examples of List Comprehensions

```
[word.lower() for word in sentence]
```

```
[word for word in sentence if len(word) > 8]
```

```
[(x, x ** 2, x ** 3) for x in range(10)]
```

Examples of List Comprehensions

```
[word.lower() for word in sentence]
```

```
[word for word in sentence if len(word) > 8]
```

```
[(x, x ** 2, x ** 3) for x in range(10)]
```

```
[(i,j) for i in range(5) for j in range(i)]
```

Examples of List Comprehensions

```
[word.lower() for word in sentence]
```

```
[word for word in sentence if len(word) > 8]
```

```
[(x, x ** 2, x ** 3) for x in range(10)]
```

```
[(i,j) for i in range(5) for j in range(i)]
```

Be careful - "simple is better than complex"

Your Turn

Your Turn

[0, 1, 2, 3] → [1, 3, 5, 7]

[3, 5, 9, 8] → [True, False, True, False]

range(10) → [0, 1, 4, 9, ..., 81]

Your Turn

[0, 1, 2, 3] → [1, 3, 5, 7]

[3, 5, 9, 8] → [True, False, True, False]

range(10) → [0, 1, 4, 9, ..., 81]

["apple", "orange", "pear"] → ["A", "O", "P"]

["apple", "orange", "pear"] → ["apple", "pear"]

["apple", "orange", "pear"] →

[("apple", 5), ("orange", 6), ("pear", 4)]

Other Comprehensions

Other Comprehensions

Dictionary Comprehensions

```
{key_func(vars):val_func(vars) for vars in iterable}
```


Other Comprehensions

Dictionary Comprehensions

```
{key_func(vars):val_func(vars) for vars in iterable}
```

```
{v:k for k, v in d.items()}
```

Other Comprehensions

Dictionary Comprehensions

```
{key_func(vars):val_func(vars) for vars in iterable}
```

```
{v:k for k, v in d.items()}
```

Set Comprehensions

```
{func(vars) for vars in iterable}
```

Other Comprehensions

Dictionary Comprehensions

```
{key_func(vars):val_func(vars) for vars in iterable}
```

```
{v:k for k, v in d.items()}
```

Set Comprehensions

```
{func(vars) for vars in iterable}
```

```
{word for word in hamlet if is_palindrome(word.lower())}
```

Comprehensions as Higher-Level Transformations

Comprehensions as Higher-Level Transformations

Usual Focus Modify individual elements.

Comprehensions Abstract transformations.

Don't say how to build a collection.

Just describe what output you want.

Comprehensions as Higher-Level Transformations

Usual Focus Modify individual elements.

Comprehensions Abstract transformations.

Don't say how to build a collection.

Just describe what output you want.

Functional Programming (Week 4) Go to the extreme!

Next Time

Next Time



Get Your Hands Dirty!

Explore Data Structures

Investigate Odd Behavior

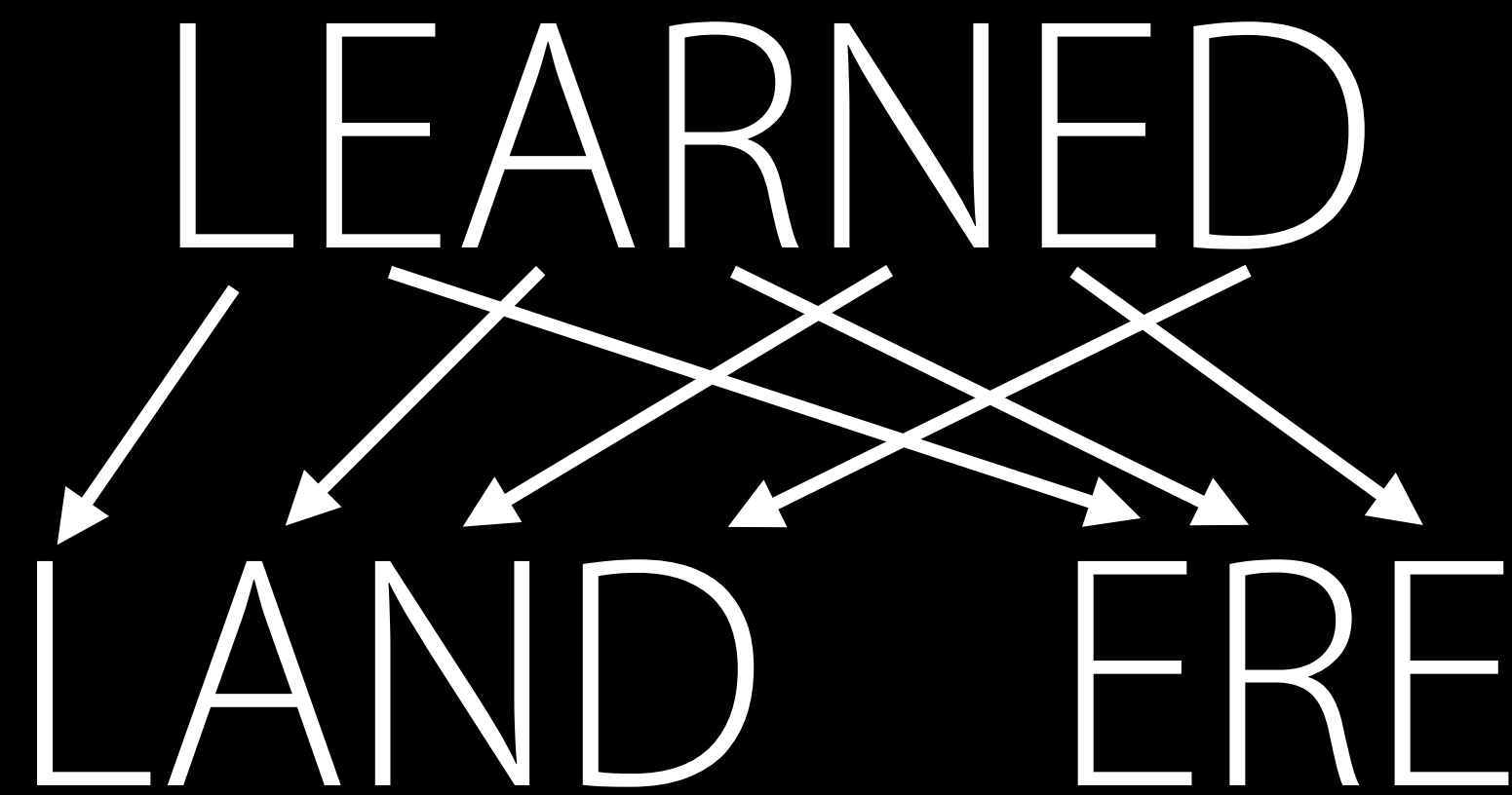


Your Turn

Triad Phrases

LEARNED THEOREM
WOODED AGRARIANS
SCHOOLED ORIOLE

Triad Phrase



Alternate letters spell out two words

Surpassing Phrases

SUPERB SUBWAY

PORKY HOGS

TURNIP FIELDS

Surpassing Phrase

SUPERB

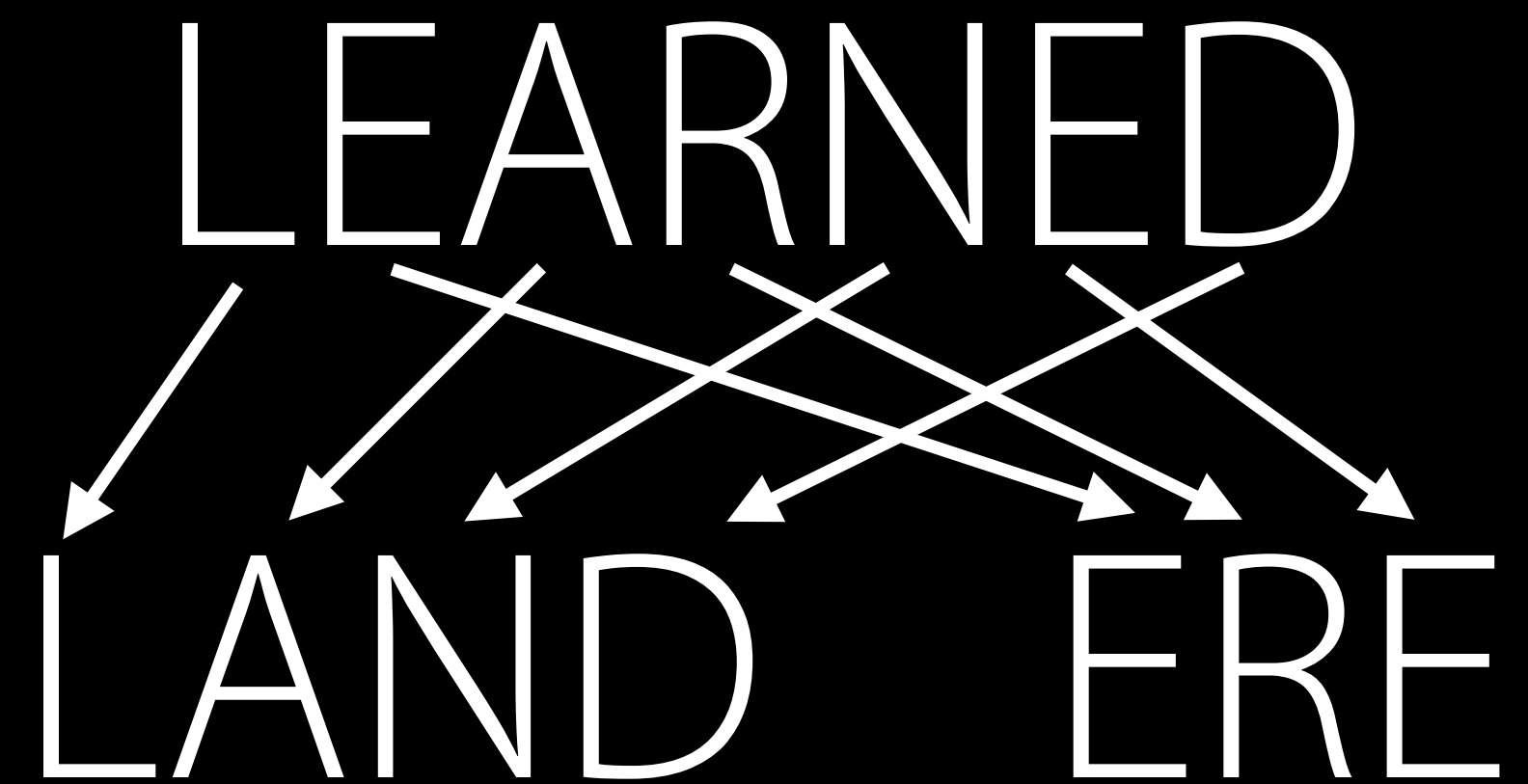
SU / UP / PE / ER / RB

2 / 5 / 11 / 13 / 16

```
ord('a') # => 97  
chr(97) # => 'a'
```

Gaps between adjacent letters increase

Triad Phrases



Surpassing Phrases

SUPERB

SU / UP / PE / ER / RB

2 / 5 / 11 / 13 / 16

```
ord('a') # => 97  
chr(97) # => 'a'
```

/usr/share/dict/words or
<http://stanfordpython.com/res/misc/words>