What is an efficient phrase?

Efficient Phrases

These are efficient…

COLD WINDOWSILL
COOL MILLION
VIVID DISILLUSIONS
SUSPICIOUS CONCLUSION

These aren't efficient…

CHILLY WINDOW LEDGE
GOOD THOUSAND THOUSAND
GRAPHIC DISAPPOINTMENTS
MISTRUSTFUL ENDING

Data Structures

January 13, 2020

Agenda

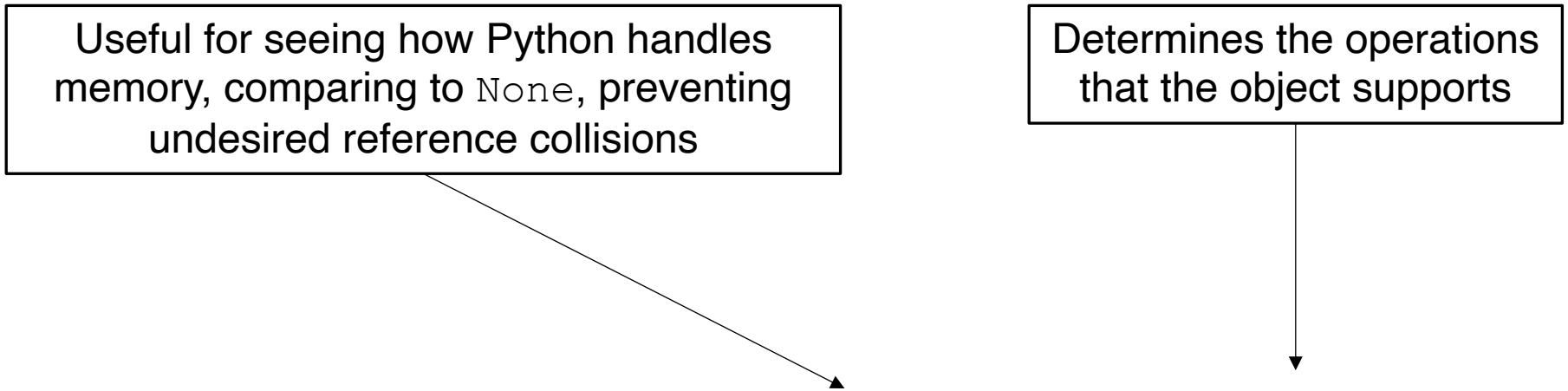
Today in CS 41

- Review
- Data Structures
- Halftime
  - Attendance
  - Setting up Python
  - Assignment 0
  - Enrollment
- More Data Structures

On to Python!*

*Follow along with the examples!

The Data Model



Objects have identity, type, and value
Variables are un-typed (dynamically typed)

Identity Crisis

```
x = 41
y = 41
x is y # => True

x = 257
y = 257
x is y # => False

x = "hap.py code"
y = "hap.py code"
x is y # => False
```

Python is “smart” and only creates one object...

But this doesn’t always happen (actually, it almost never happens)!

Maintaining only one object would require Python checking existing objects every time it considered creating a new one – too much work.

If you’re interested in more details about why Python behaves “smartly” for the number 41 but not for 257, see me after class!

The Object of Investigation

Takeaways:

Python objects are more than just their value, but you'll almost never care about the *precise* identity of an object.

Python handles the creation of new objects.

Object reassignment does not create a new object.

```
x = # ...
y = x
```

We'll see more in lab!

Strings

Useful String Methods

```
greeting = "Hello! Love, unicorn.  "

greeting[4]           # => 'o'
'corn' in greeting    # => True
len(greeting)         # => 23

greeting.find('lo')                  # => 3 (-1 if not found)
greeting.replace('ello', 'iya')      # => Hiya! Love, Unicorn.
greeting.startswith('Hell')          # => True
greeting.endswith(' ')               # => True
greeting.isalpha()                   # => False
```

Useful String Methods

```
greeting = "Hello! Love, unicorn.  "

greeting.lower()          # => 'hello! love, unicorn.'
greeting.title()          # => 'Hello! Love, Unicorn.'
greeting.upper()          # => 'HELLO! LOVE, UNICORN.'
greeting.strip()          # => 'Hello! Love, unicorn.'
greeting.strip('.nrH ') # => 'ello! Love, unico'
```

Lists <—> Strings

```
list('Hair toss!')
# => ['H', 'a', 'i', 'r', ' ', 't', 'o', 's', 's', '!']

# `.split` partitions by a delimiter...
'ham cheese bacon'.split()
# => ['ham', 'cheese', 'bacon']

# ...which can be specified, but defaults to whitespace
'3-14-2015'.split(sep='-')
# => ['3', '14', '2015']

# `.join` creates a string from a list of strings
', '.join(['Zheng', 'Antonio', 'Sam'])
# => 'Zheng, Antonio, Sam'
```

String Formatting

```
# Curly braces are placeholders
'{} {}'.format('beautiful', 'unicorn') # => 'beautiful unicorn'

# Provide values by position or placeholder
'{0} can be {1} {0}, even in summer!'.format('snowmen', 'frozen')
# => 'snowmen can be frozen snowmen, even in summer!'

'{name} loves {food}'.format(name='Michael', food='applesauce')
# => 'Michael loves applesauce' (he does)

# Values are converted to strings
'{} squared is {}'.format(5, 5**2) # => '5 squared is 25'
```

String Formatting

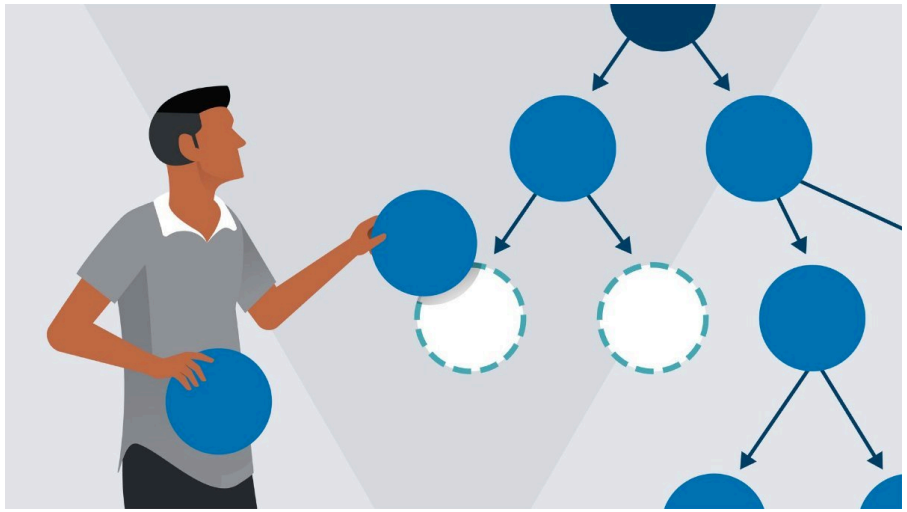
```
# You can use C-style specifiers too!
"{:06.2f}".format(3.14159) # => '003.14'

# Padding can be specified as well.
'{:10}'.format('left') # => 'left      '
'{:*^12}'.format('CS41') # => '****CS41****'

# You can even look up values!
captains = ['Kirk', 'Picard']
"{caps[0]} > {caps[1]}".format(caps=captains)
```

See https://pyformat.info/ for more!

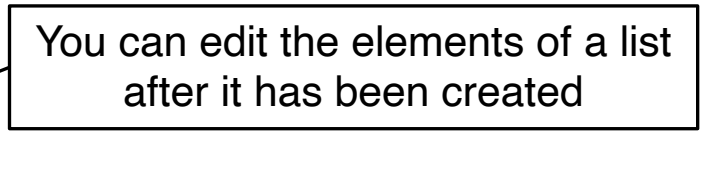
Data Structures



- Lists
- Tuples
- Dictionaries
- Sets
- Advanced Looping
- Comprehensions

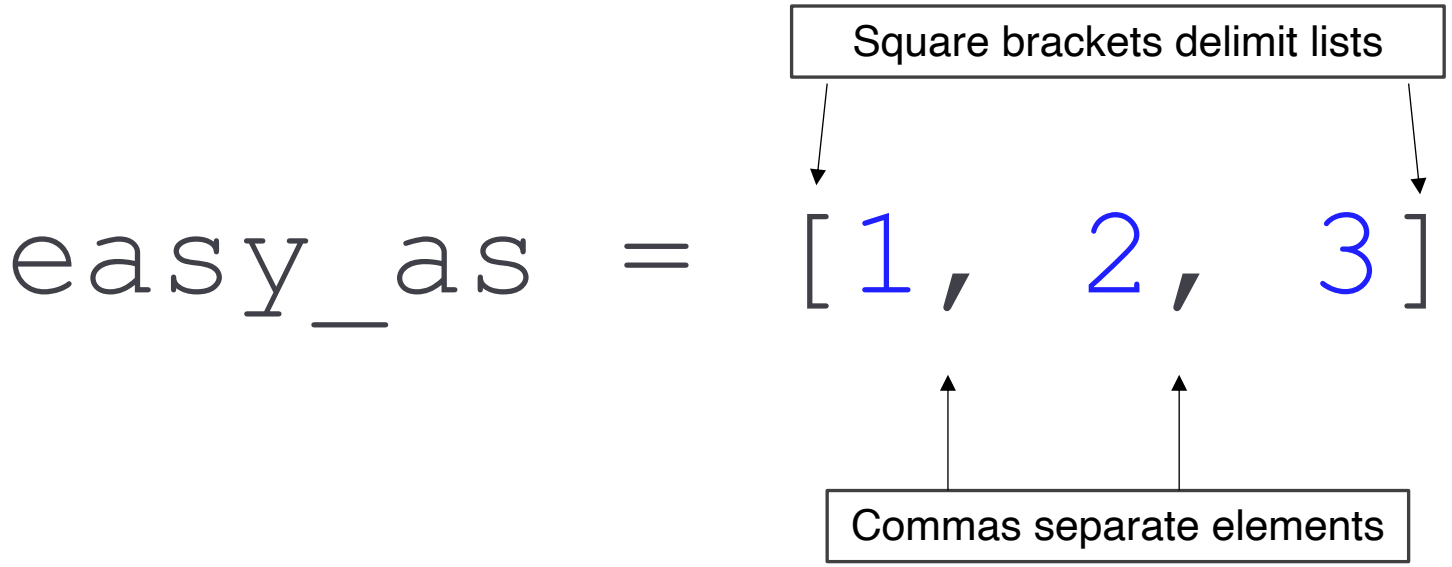
Lists

What is a list?



Finite, ordered, mutable sequence of elements.

What is a list?



What is a list?

```
simple_as = ['do', 're', 'mi']
```

Adding Elements

```
# 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 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

```
# You can put anything inside a list!

# 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'
```

Recursive Lists

```
# You can put anything inside a list!

# the list itself...
x = [1, 2]
x.append(x)
x               # => [1, 2, [...]]
x is x[2]       # => True
```

Iterable Methods

```
# Length (number of objects in the iterable)
len([])                    # => 0
len("")                    # => 0
len("unicorn")             # => 7
len([4, 5, "seconds"])     # => 3
```

Iterable Methods

```
# Membership (in)
"u" in "unicorn"          # => True
"uni" in "unicorn"        # => True
"uncorn" in "unicorn"     # => False
4 in [4, 5, "seconds"]    # => True

# Casting to a list
list("unicorn")   # => ['u', 'n', 'i', 'c', 'o', 'r', 'n']
list(range(4, 6))# => [4, 5]
```

List-Specific Methods

```
# Count the number of occurrences in a list
my_lst.count(value)

# Append a *single* element to a list
my_lst.append(elem)

# Extend a list by appending *all* elements from an
iterable
my_lst.extend(iterable)
```

List-Specific Methods

```
# Insert an element into the middle of the list, at index
my_lst.insert(index, elem)

# Sort a list, in place
my_lst.sort(key=None, reverse=False)

# Reverse a list, in place
my_lst.reverse()
```

List-Specific Methods

```
# Modify multiple elements from a list
# if len(iterable) == len(my_lst[start:stop:step])...
my_lst[start:stop:step] = iterable
del my_lst[start:stop:step]

# Remove an element from a list and return it
my_lst.pop()  # => returns & removes last elem
my_lst.pop(i) # => returns & removes my_lst[i]
```

List-Specific Methods

```
# Remove an element from a list, by value
# removes the first instance or throws ValueError
my_lst.remove(value)
```

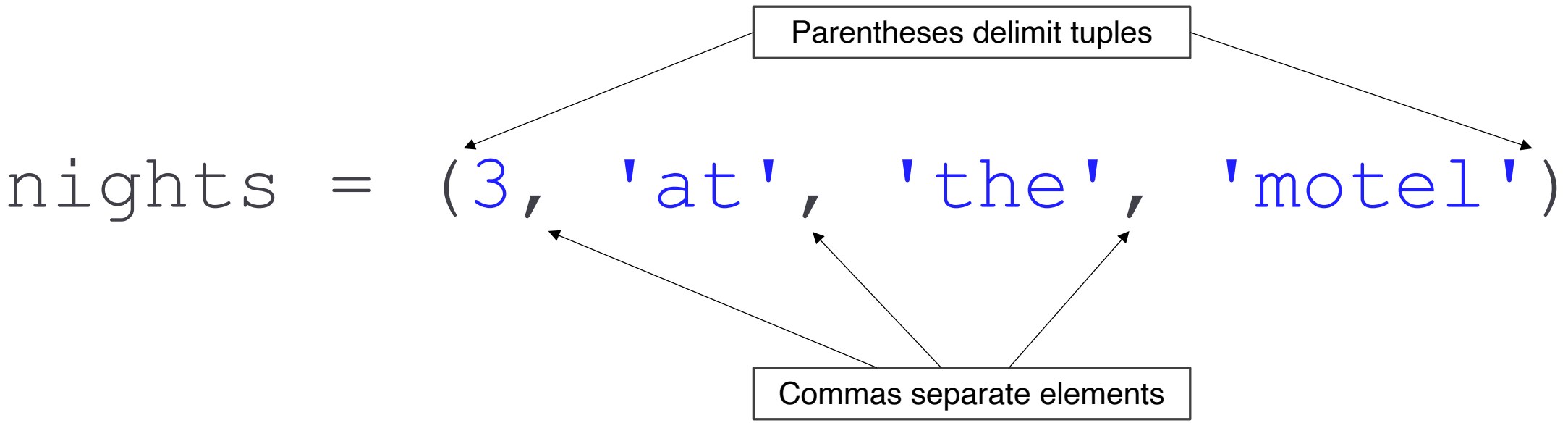
Tuples

What is a tuple?

You **can't** edit the elements of a tuple after it's been created (different from a list)

Finite, ordered, immutable sequence of elements.

What is a Tuple?



But… why?

We already have lists, so why tuples?

Store finite, heterogenous data about an object in our program.

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

“Freeze” a sequence for hashability (lists can’t be hashed)

Enforce immutability

Working with Tuples

```
# You can't edit elements...
fish = (1, 2, "red", "blue")
fish[0]                 # => 1
fish[0] = 'unicorn' # TypeError

# ...but everything else works normally
len(fish)           # => 4
fish[:-1]           # => (1, 2, 'red')
"red" in fish   # => True
```

Packing & Unpacking Tuples

```
christopher_robin = 'pooh', 'tigger', 'eeyore'
print(christopher_robin) # ('pooh', 'tigger', 'eeyore')
type(christopher_robin)  # => tuple

# We can unpack these values!
x, y, z = christopher_robin
x # => 'pooh'
y # => 'tigger'
z # => 'eeyore'
```

Swapping Values

```
riri = "Sorry, I ain't sorry"
bey = "Come Mr. DJ won't you turn the music up"
```

```
temp = riri
riri = bey
bey = temp
```

Temporary variable

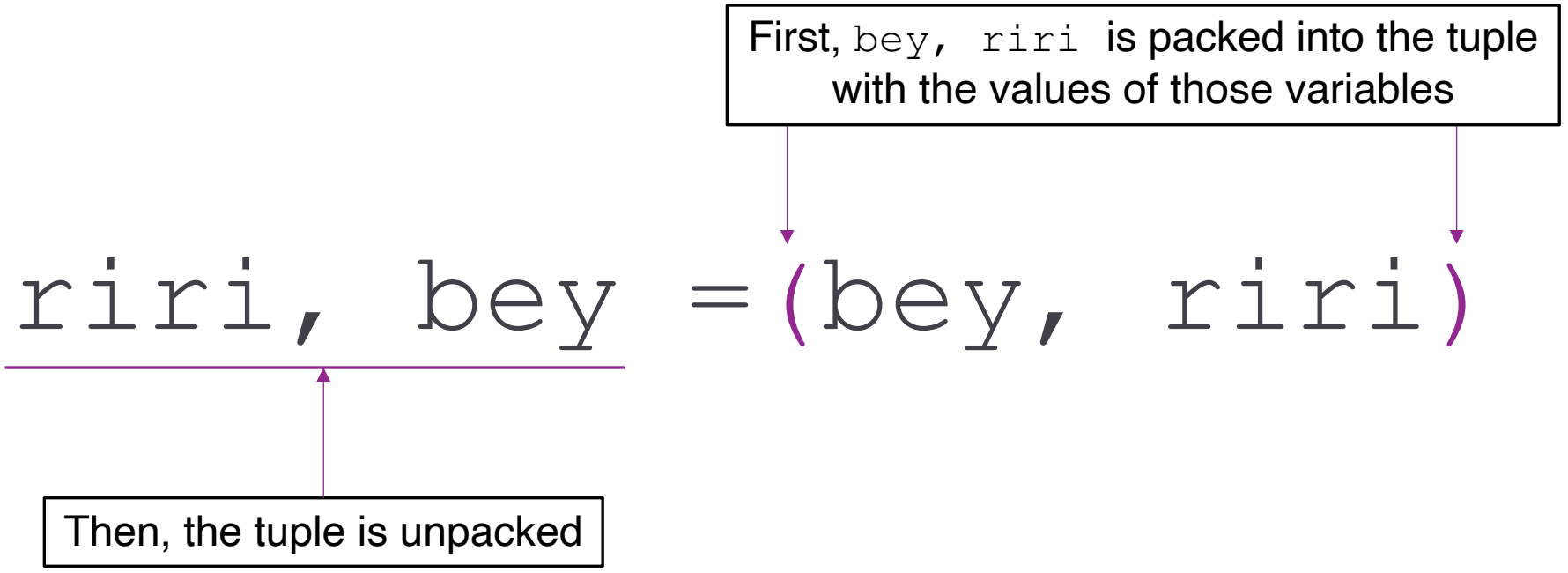
```
riri = riri ^ bey
bey = riri ^ bey
riri = riri ^ bey
```

Bitwise XOR magic

```
riri, bey = bey, riri
```

Tuple unpacking…!

Swapping Values: How does it work?



Fibonacci Sequence

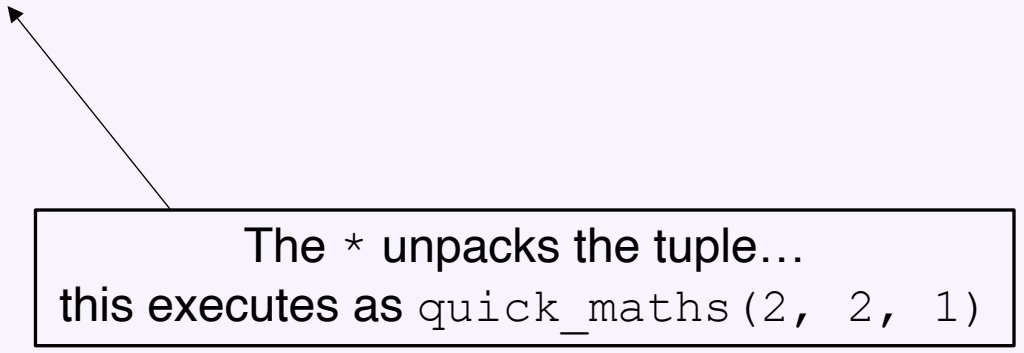
```
def fibbi(n):
    """Print out the first n Fibonacci numbers"""
    a, b = 0, 1

    for i in range(n):
        print(a)
        a, b = b, a + b
```

Packing & Unpacking Tuples into Functions

```
def quick_maths(a, b, c):
    return a + b - c

boom = (2, 2, 1)
quick_maths(*boom)  # => 3
```



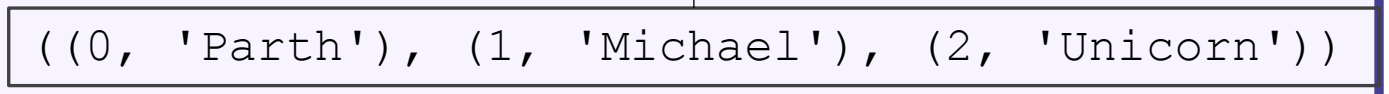
A Useful Unpacking Paradigm

```
enumerate(['Parth', 'Michael', 'Unicorn'])
# => ((0, 'Parth'), (1, 'Michael'), (2, 'Unicorn'))
```

A Useful Unpacking Paradigm

```
for i, name in enumerate(['Parth', 'Michael', 'Unicorn']):
    print(i, name)

# 0 Parth
# 1 Michael
# 2 Unicorn
```



A Quirk

```
org = ("Dumbledore's Army", ['harry', 'hermione', 'ron'])
org[1].append('cho')

org
# => ("Dumbledore's Army", ['harry', 'hermione', 'ron',
'cho'])
```

Tuples contain immutable *references* to underlying objects

If those objects are not hashable, the tuple will not be hashable either

Announcements @ Halftime

Setting Up Python

- Python 3.8.0
- Using Virtual Environments
- Detailed instructions in Assignment 0.
- Video Tutorial here.
- Stuck?
  - See us after or come to OH!
  - Post on Piazza
  - Say “Python” three times to the mirror at night… Michael will appear.

Assignment 0

- Set up Python and get into the basic flow
- Due **11:59pm, Tuesday Jan 21**
  - *Please* send us an email if you need anything! We don't want this to be stressful.
  - Email Parth and Michael (psarin@stanford.edu & coopermj@stanford.edu) or your TA.
- Submission via Paperless (see the CS41 Submission Instructions for more details)

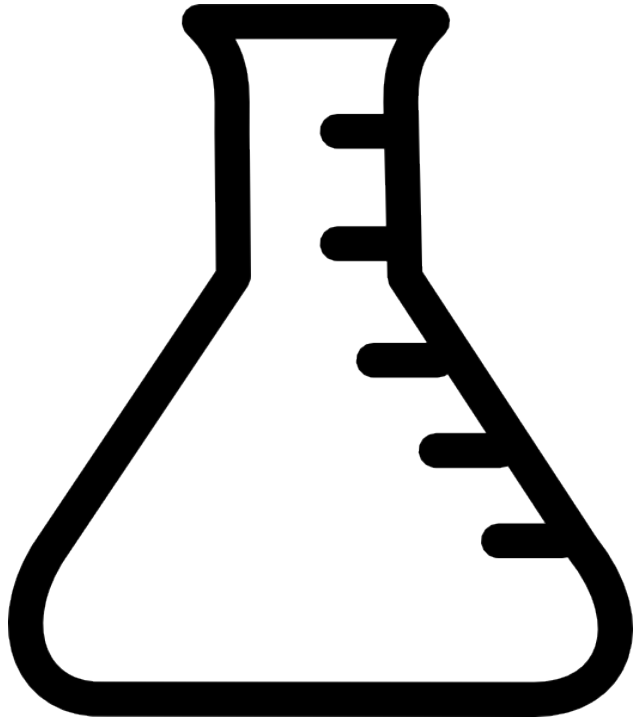
Enrollment

If you have not already received an enrollment code, and have confirmed that you can enroll in this class:

https://stanfordpython.com/admissions.html

(There's also an announcement about this in the course news feed)

Lab!



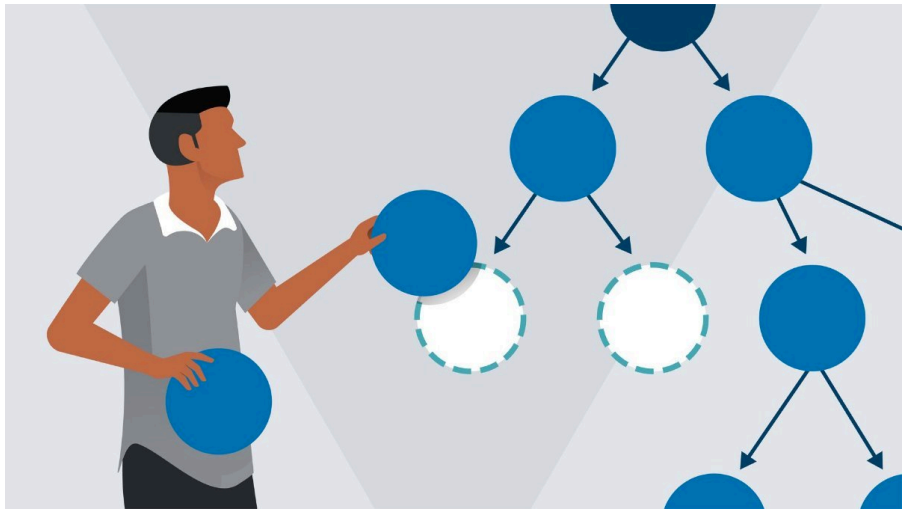
Fundamentals and More

Lab Groups with Course Staff

Bring a charged computer!

Back to today's topics…

Data Structures



- Lists
- Tuples
- Dictionaries
- Sets
- Advanced Looping
- Comprehensions

Dictionaries

What is a Dictionary?

Keys can be any *hashable* type

Mutable map from hashable values to (arbitrary) objects

Working with Dictionaries

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

a = {"one": 1, "two": 2, "three": 3}
b = dict(one=1, two=2, three=3)
c = dict([('one', 1), ('two', 2), ('three', 3)])
a == b == c # => True
```

Access and Mutate

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

# Access
d['one']  # => 1
d['five'] # KeyError

# Mutate
d['two'] = 22  # Modify an existing key
d['four'] = 4  # Add a new key
```

Get (with Default)

```
d = {"CS": [41, 106, 107], "LAW": [4093, 7007]}
d["PHIL"]      # KeyError

d.get("CS")    # => [41, 106, 107]
d.get("PHIL") # => None (not a KeyError!)

# But what if None is a valid entry in your dict?
english_classes = d.get("ENGLISH", [])
num_english = len(english_classes)
```

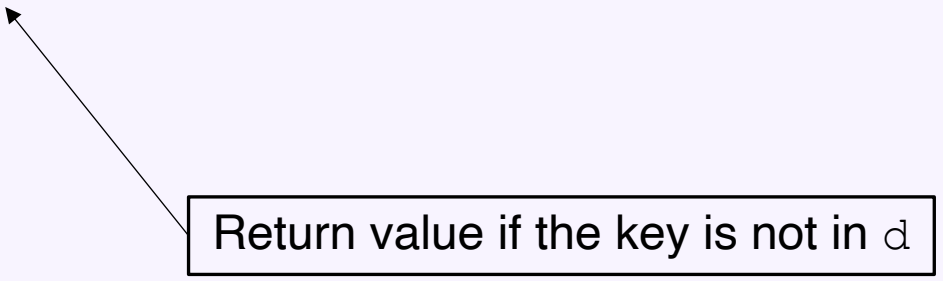
Return value if the key is not in `d`

Deleting Elements

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

del d['one']
del d['five'] # KeyError

d.pop("three", None) # => 3
```



Dictionary Structure

```
d = {"one": 1, "two": 2, "three": 3}
d.keys()   # => <"one", "two", "three">
d.values() # => <1, 2, 3>
d.items()  # => <('one', 1), ('two', 2), ('three', 3)>

for key, value in d.items():
    print(key, value)

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

Insertion *in order*. What would `{'a': 1, 'a': 2}` evaluate to?

Dynamic references to the keys, values, and items of the dictionary

Common Operations

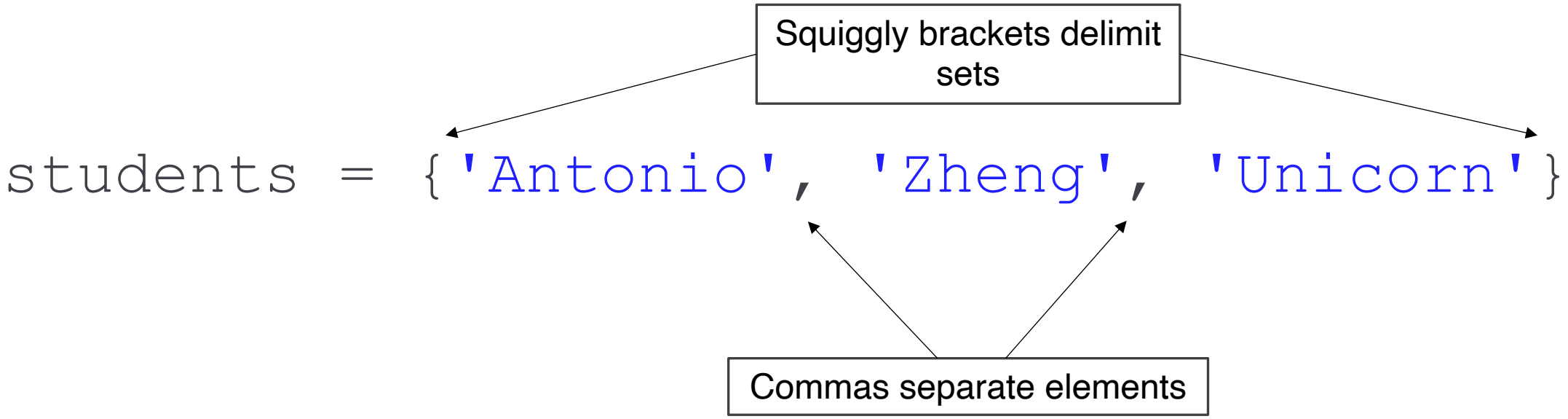
```
len(d)              # number of keys
key in d            # equiv to `key in d.keys()`
value in d.values()
d.copy()            # makes a shallow copy
d.clear()
for key in d: # equiv to `for key in d.keys()`
    print(key)
```

Sets

What is a set?

Unordered, finite collection of distinct, hashable elements.

What is a set?


Squiggly brackets delimit sets
students = {'Antonio', 'Zheng', 'Unicorn'}
Commas separate elements

But… why?

We already have lists and tuples, so why sets?

Fast membership testing

- Using hashing, membership is O(1) vs. O(n)

Eliminate duplicate entries (O(n) to eliminate duplicates from a list)

Easy set operations (intersection, union, etc.)

Common Set Operations

```
empty_set = set() # why not {}?
set_from_list = set([1, 2, 1, 4, 3]) # => {1, 3, 4, 2}

basket = {'orange', 'banana', 'pear', 'apple'}
len(basket)                  # => 4
'orange' in basket           # => True
'crabgrass' in basket        # => False

for fruit in basket:
    print(fruit, end=" 🤠 ")
# pear 🤠 apple 🤠 banana 🤠 orange 🤠
```

O(1) membership testing

Common Set Operations

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

a.add('unicorn')
a.remove('m')  # KeyError if 'm' is not in a
a.discard('x') # same as remove, but no error

a.pop()    # => 's' (or 'i' or 'p' or 'unicorn')
a.clear() # removes all elements
```

Common Set Operations

```
a = set('abracadabra') # {'a', 'r', 'b', 'c', 'd'}
b = set('alacazam')    # {'a', 'm', 'c', 'l', 'z'}
# Set difference (letters in a but not in b)
a - b # => {'r', 'd', 'b'}
# Set union (letters in a or b or both)
a | b # => {'a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'}
# Set intersection (letters in both a and b)
a & b # => {'a', 'c'}
# Symmetric difference (letters in a or b but not both)
a ^ b # => {'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'}
# Update a set's value with any of the operators
a |= b

# Set containment (subsets and supersets)
a < b  # strict subset
a <= b # subset or equal to
a > b  # strict superset
a >= b # superset or equal to
```

Coding `is_efficient`

```
EFFICIENT_LETTERS = 'BCDGIJLMNOPSUVWZ'

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

Coding `is_efficient`

```
EFFICIENT_LETTERS = set('BCDGIJLMNOPSUVWZ')

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

Advanced Looping

zip

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


# What is your name? It is lancelot.
# What is your quest? It is the holy grail.
# What is your favorite color? It is blue.
```

Generates pairs of entries from its arguments

sorted

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

# apple
# banana
# orange
# pear
```

Returns a new sorted list while leaving the source unaltered

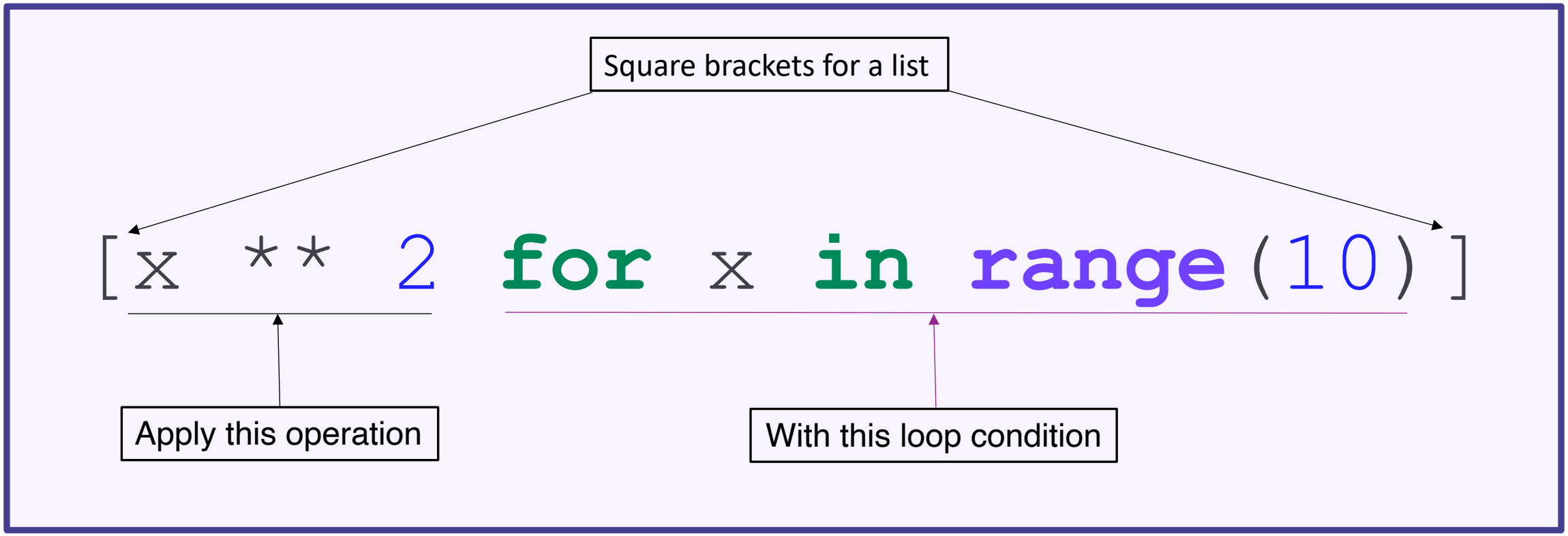
Comprehensions

Square Numbers

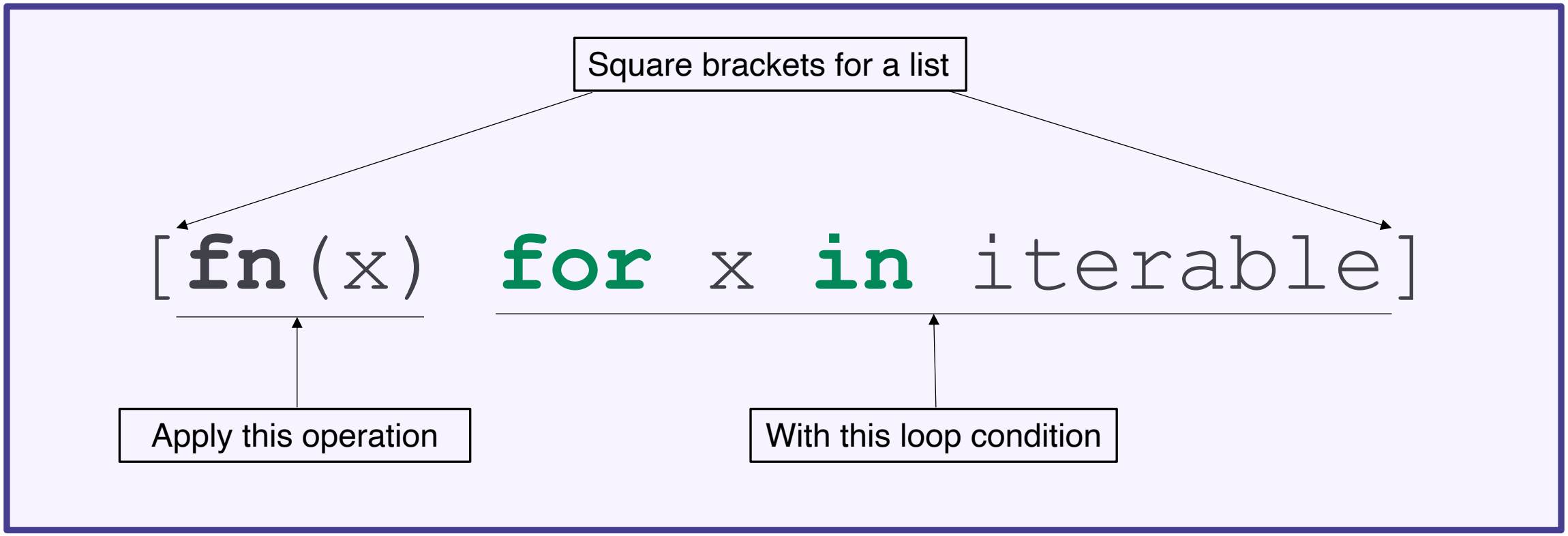
```
squares = []
for x in range(10):
    squares.append(x**2)

squares # => [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Square Numbers (Comprehension)



List comprehensions



List comprehensions

```
[fn(x) for x in iterable if cond(x)]
```

Only keep elements that satisfy a boolean condition

Examples

```
[word.lower() for word in sentence]
[ch for ch in word.lower() if ch not in 'aeiou']

[(x, x**2, x**3) for x in seq]
[(i,j) for i in range(5) for j in range(i)]
```

Be careful! Simple is better than complex.

Your turn!

```
[0, 1, 2, 3] -> [1, 3, 5, 7]
[3, 8, 9, 5] -> [True, False, True, False]

['apple', 'orange', 'pear'] -> ['A', 'O', 'P']
['apple', 'orange', 'pear'] ->
    [('apple', 5), ('orange', 6), ('pear', 4)]
```

Other Comprehensions

```
# Dictionary comprehensions
{key_fun(x):val_fun(x) for x in iterable}
fav_animals = {
    'parth': 'unicorn',
    'michael': 'elephant',
    'zheng': 'tree',
    'sam': 'ox',
    'nick': 'Daisy'
}
fav_humans = {val:key for key, val in fav_animals.items()}

# Set comprehensions
{fun(x) for x in iterable}
```

A High-Level View

Why would we use comprehensions?

Usual focus: Modify individual elements of an iterable.

Comprehensions ~ Abstract Transformations

Don't say how you want the object constructed.

Just say what you want *in* the object.

If you liked this, stay tuned… Functional Programming!

Next time…

Agenda

Wednesday in CS 41:

Get your hands dirty with our first lab!

- Confirm that Python is running correctly.
- Add some fun tracks to our playlist!

Explore data structures

- Use them to solve interesting problems!
- See how Python handles them behind the scenes (and investigate odd behavior).

Your TAs will reach out to you about your lab group assignment shortly!