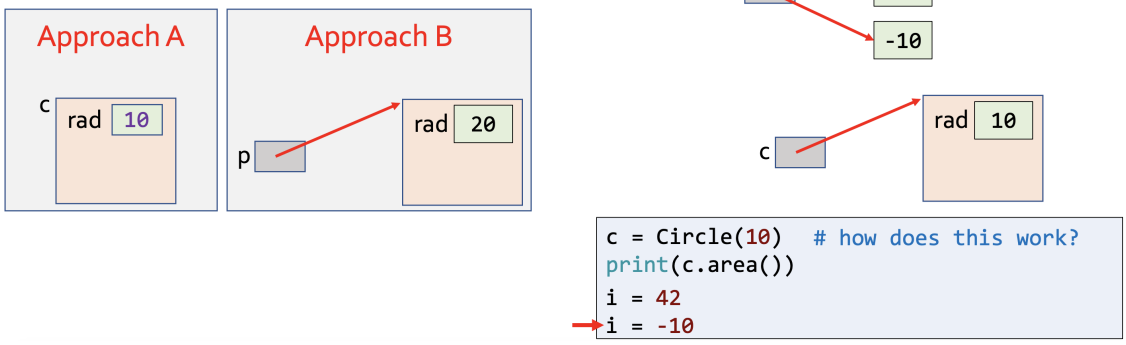
3 - Python

Variables

- only makes `__name__` var when `main()` present to call `__name__==__main__`
- variables have scope within functions, not within conditional blocks
- double underscore to make member functions private
- ALL data is object references, and all of it is stored in a heap - inefficient
  - data on the heap is immutable, instead, it creates a new object and references that and garbage collects the old



- data allocation size is managed by a dictionary that determines the size

Classes

- class variables/objects (non self, just defined at the top of the class) are accessible by all objects of that class

thing.py

```
class Thing:
 thing_count = 0

 def __init__(self, v):
  self.value = v
  Thing.thing_count += 1
  self.thing_num = Thing.thing_count
```

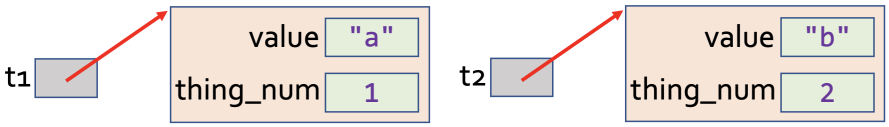
Thing class

thing_count 2

__init__(self, v)

method code...

```
from thing import Thing

t1 = Thing("a")
print(f"{t1.thing_num} {Thing.thing_count}")

t2 = Thing("b")
print(f"{t1.thing_num} {Thing.thing_count}")
print(f"{t2.thing_num} {Thing.thing_count}")
```



Challeng
following p

```
1 1
1 2
2 2
```

Let

Only use a cla
need a variable
across all of yo

- class methods have no `self` param and cannot access class member variables

thing.py

```
class Thing:
 thing_count = 0

 def __init__(self, v):
  self.value = v
  Thing.thing_count += 1
  self.thing_num = Thing.thing_count

 def change_val(self, new_val):
  self.value = new_val

 def a_class_method(foo, bar):
  return Thing.thing_count * foo + bar

 def another_class_method(bletch):
  Thing.a_class_method(bletch,20)
```

```
from thing import Thing

t1 = Thing("a")
Thing.another_class_method(42)
```

A class method is a method that has no self parameter. As such...

- it can't access member variables
- it can't call instance methods

So what can they do?

They can...

- access class variables
- call other class methods

To call a class method, use this syntax:
class_name.method_name(params)

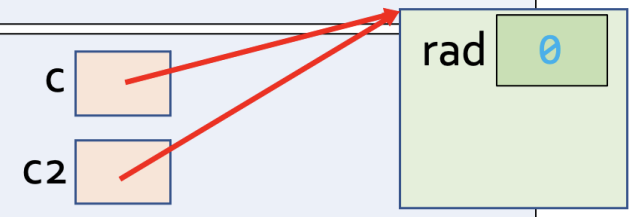
Use class methods when the only state that needs to be operated on is class variables, or the method does not need any not state.

Copying

- copying is by default soft copy and creates object references

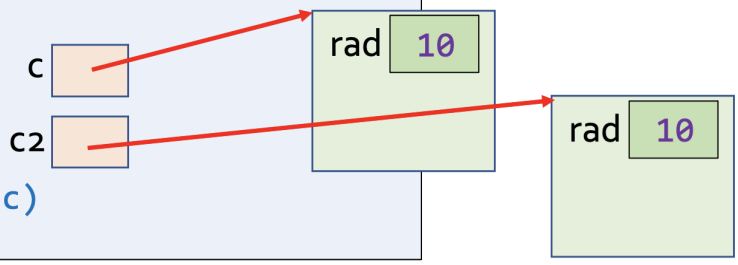
```python
# Circle class in Python
import math

class Circle:

  def __init__(self, rad):
    self.rad = rad

  def area(self):
    return math.pi * self.rad**2

  def set_radius(self, rad):
    self.rad = rad
```

```python
c = Circle(10)
print(c.area())
c2 = c
c.set_radius(0)
print(c2.area())  # prints 0
```



- use `copy.deepcopy` does a recursive, depth-first copy of the actual object, not just the reference

```python
import copy

c = Circle(10)
print(c.area())

c2 = copy.deepcopy(c)
```

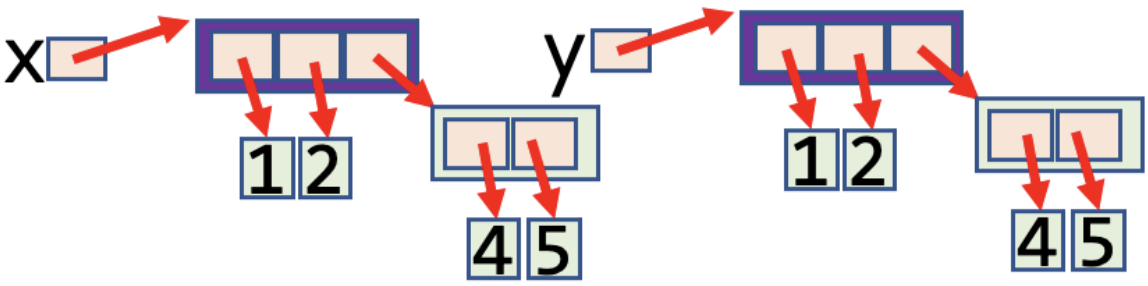


- shallow copy (`copy.copy`) only copies the top level object (not recursive) i.e. in the example the pointers in thee array point to the same values as the other array

Python has deep copying:

A deep copy makes a copy of the top-level object *and* every object referred to directly or indirectly by the top-level object:

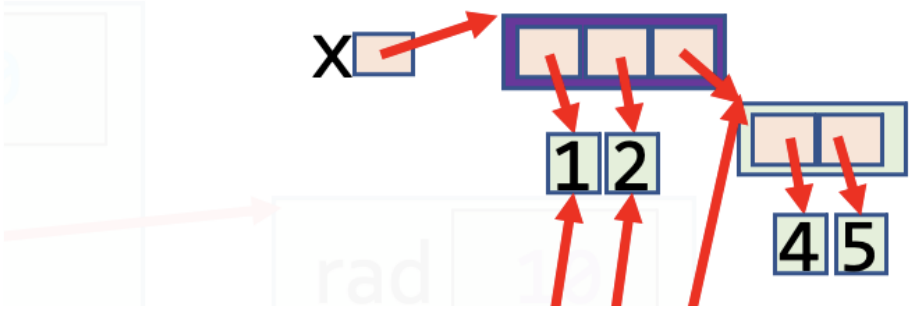
y = copy.deepcopy(x)

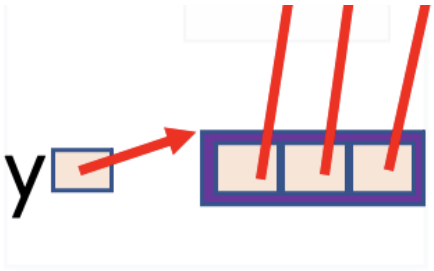


Python has shallow copying:

A shallow copy just makes a copy of the top level object:

y = copy.copy(x)





Garbage collection

- garbage collection is automatic
- destructor defined using `__del__` but not guaranteed to run and rarely used
- finalizer runs before the garbage collector to finalize anything with that object, then garbage collects i.e. destructor but is being phased out
- instead, define your own disposal method to dispose

Inheritance

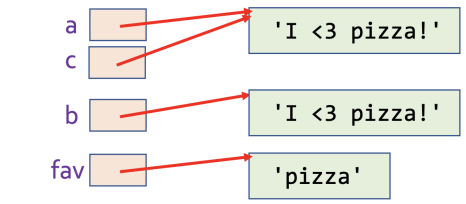
```
class Person:
  def __init__(self, name):
    self.name = name

  def talk(self):
    print('Hi!\n')




class Student(Person):
  def __init__(self, name):
    super().__init__(name)
    self.units = 0

  def talk(self):
    print(f"Heya, I'm {self.name}.")
    print("Let's party! Oh... and ")
    super().talk()
```

```
def chat(p):
 p.talk()

def cs131_lecture():
 s = Student('Angelina')
 chat(s)
```

Objects

Object Equality

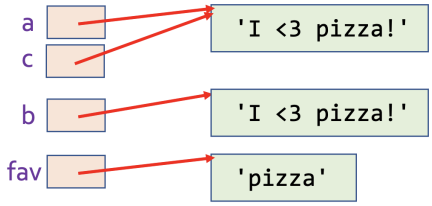
```python
# Different types of equality in Python
fav = 'pizza'
a = f'I <3 {fav}!'
b = f'I <3 {fav}!'
c = a

if a == b:
  print('Both objects have same value!')
if c is a:
  print('c and a refer to the same obj')
if a is not b:
  print('a and b refer to diff. objs')
```




```
Both objects have same value!
c and a refer to the same obj
a and b refer to diff. objs
```

Object IDs

```python
# Different types of equality in Python
fav = 'pizza'
a = f'I <3 {fav}!'
b = f'I <3 {fav}!'
c = a

print(id(a))
print(id(b))
print(id(c))
```




```
140426129935136
140426129649376
140426129935136
```

None

- acts like a `nullptr`

```python
q = None

if q is False:
  print('Is None the same as False?')

if not q:
  print('Does not work with None?')

if q is None:
  print('Ahhh q is None!')

if q == None:
  print('Ahh q == None!')
```

Challenge: What will the program the left print?

```
Does not work with None?
Ahh q is None!
Ahh q == None!
```

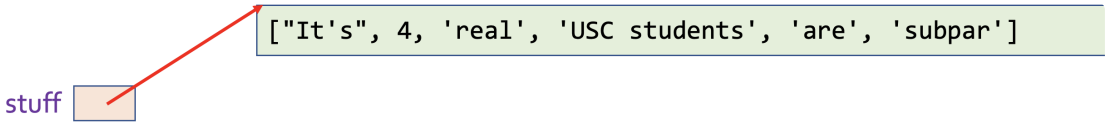
Strings

- Strings are immutable, just like all other objects

- treated as a list of chars, so use list indexing to access substrings

Lists - MIDTERM

- lists ARE mutable



Challenge: What will this program print?

```
stuff = [42, False, 'walnuts']

stuff[2] = 'USC students'
stuff[0:2] = ["It's", 4, 'real']
stuff.append('are')    # or stuff += ['are']

stuff = stuff + ['subpar']
print(stuff[3:])
if 'walnuts' not in stuff:
  for s in stuff:
    print(s)
```

```
['USC students', 'are', 'subpar']
It's
4
real
USC students
are
subpar
```

- `a.append(b)` == `a+=b` ≠ `a = a+b`
- List implementation: array of object references

Challenge: How do you think lists are implemented in Python?

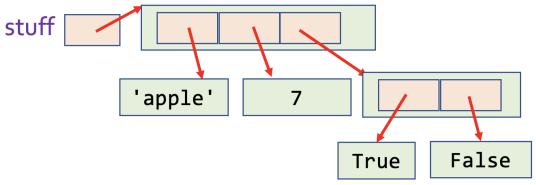
Based on this, what's the big-O of:

Using the "in" and "not in" commands — O(n) for a list of size n

Accessing list[j] where j is large — O(1)

Appending one list onto another — O(m+n)

```
# Simple list in Python

stuff = ['apple', 7, [True, False]]
```



Tuples

- immutable ordered groups

```python
def get_school_and_scholarship(gpa):
  if gpa > 4.2:
    return ('UCLA', 0)
  else:
    return 'USC', 100000


tup = get_school_and_scholarship(4.5)
print(f'You got into {tup[0]} with ${tup[1]}')

skool, dough = get_school_and_scholarship(1.3)
print(f'You got into {skool} with ${dough}')
```

```
You got into UCLA with $0
You got into USC with $100000
```

-

Sets

- Stores a single unique copy
- not ordered alphabetically
- implemented using hash tables

```
draining = set()

draining.add('studying')
draining.add('CS131')
draining.add('dating')
draining.add('studying')
draining.remove('CS131')
print(draining)

if 'CS131' not in draining:
  print('Studying for CS131 is NOT draining!')

# Let's create a set from a list...
dinner = ['salad','soup','steak','soup','pie']
dinner_set = set(dinner)
print(f'Unique foods: {dinner_set}')
```

```
{'dating', 'studying'}
Studying for CS131 is NOT draining!
Unique foods: {'soup', 'steak', 'pie', 'salad'}
```

- sets have simple, built in set operations

```python
# Common set operations are supported by Python

adolescence = {10,11,12,13,14,15,16,17,18,19}
moody_ages = {13,14,15,16,17}

pleasant_yrs = adolescence - moody_ages
print(pleasant_yrs)

fruits = {'apple', 'tomato'}
veggies = {'kale', 'tomato'}

healthy = fruits | veggies
print(healthy)

common = fruits & veggies
for fv in common:
  print(fv)
```

```
{10, 11, 12, 18, 19}
{'kale', 'tomato', 'apple'}
```

Parameter Passing

- by object reference

```
def nerdify(s):
 s = 'coding ' + s

i_like = 'parties'
nerdify(i_like)
print(i_like)
```

```
parties
```

```
def peachify(f):
 f = f + ['peach']

fruits = ['apple', 'cherry']
peachify(fruits)
print(fruits)
```

```
['apple', 'cherry']
```

```
def peachify2(f):
 f.append('peach')

fruits = ['apple', 'cherry']
peachify2(fruits)
print(fruits)
```

```
['apple', 'cherry', 'peach']
```


```
def largeify(c):
 c = Circle(10)

unit = Circle(1)
largeify(unit)
print(unit.radius())
```

```
1
```

```
def largeify2(c):
 c.set_radius(10)

unit = Circle(1)
largeify2(unit)
print(unit.radius())
```

```
10
```

Error Handling

- called exceptions
- python provides a stack traceback to track where the error came from
- use try/except block to handle exceptions
- you can except for all or per error or as some variable `e`

```
def div(a, b):
  temp = a/b
  return temp

def main():
  try:
   result = div(10, 0)
   print(f'The result was {result}')
  except ZeroDivisionError:
   print('You divided by zero!')
  except TypeError:
   print('Incompatible types!')

main() # call main function
```

```
try:
        # some code
except Exeption as e:
        print("There was a",e)
```

Multi-Threading

- Python starts the first thread and it gains exclusive access to the objects

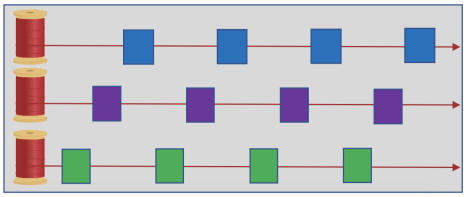
```
import threading

def task():
  n = 0
  while n < 100000000:   # does some computation
    n = n + 1

print("Creating threads!")
t1 = threading.Thread(target=task)
t2 = threading.Thread(target=task)
t3 = threading.Thread(target=task)

print("Start executing threads!")
t1.start()
t2.start()
t3.start()

print("Waiting for each thread to finish!")
t1.join()
t2.join()
t3.join()

print("All threads have finished!")
```

Assuming looping 100 million times takes 5s, how long will this program take to run on a multicore PC?

We'd expect all three tasks to run in parallel... taking 5s total. But it takes 15s!

Why? Because when each Python thread runs, it claims exclusive access to Python's memory/objects.

So only one thread generally does computation at a time!



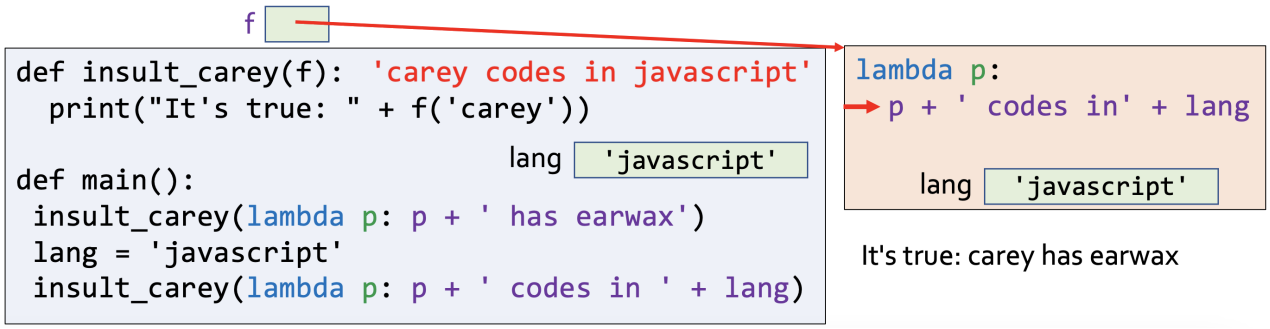
- instead, Python has a GIL (Global Interpreter Lock) i.e., a mutex (lock), so each thread releases the GIL when its done with it access
- instead, multi-threading is for non-computational, e.g. I/O (Reading/writing from disk) and libraries where they are written in C/C++

Comprehensions

```
numz = [x**2 for x in range(3,6)]
print(numz)

words = ['onomatopoeia','goober','incognito','lit']
wordz1 = [w for w in words if len(w) > 6]
print(wordz1)

s = "David's dirty dog drank dirty water down by the dam"
wordz2 = [w for w in s.split() if w[0] == 'd']
print(wordz2)

wordz3 = {w for w in s.split() if w[0] == 'd'} # hint: set
print(wordz3)

wordz4 = {w:len(w) for w in s.split()}
print(wordz4)
```

```
[9, 16, 25]
['onomatopoeia', 'incognito']
['dirty', 'dog', 'drank', 'dirty', 'down', 'dam']
{'drank', 'dam', 'dirty', 'down', 'dog'}
{"David's": 7, 'dirty': 5, 'dog': 3, 'drank': 5,
'water': 5, 'down': 4, 'by': 2, 'the': 3, 'dam': 3}
```

Lambdas



It's true: carey has earwax