Classes + Memory
Chris Piech and Mehran Sahami
CS106A, Stanford University
Remember this?
Bouncing Balls
Hope you are all doing well
Learning Goals

1. Practice with classes
2. See how to trace memory with classes
Guiding question for today:

what does it take to go from what you know to writing big-scale software?
Some *large* programs are in Python
How?
Define New Variable Types

Song

Playlist

User

Song Player

Song Retriever

Piech + Sahami, CS106A, Stanford University
You Have Been *Using* Variable Types

SimpleImage

Canvas

Karel

String

int

What would it take to define your own?
type
Classes define new variable *types*
Classes decompose your program across files
**class**: A template for a new type of variable.

A blueprint is a helpful analogy.

When defining a new variable type you make a blueprint.

Piech + Sahami, CS106A, Stanford University
You must define three things

**Instance Variables**
1. What **sub-variables** does each instance store?

**Instance Methods**
2. What **methods** can you call on an instance?

**Constructor**
3. What happens when you make a **new** one?

*details on how to define these three things coming soon*
Classes Review

dog.py

class Dog:
    def __init__(self):
        self.times_barked = 0
    def bark(self):
        print('woof')
        self.times_barked += 1

life.py

def main():
    simba = Dog()
    juno = Dog()
    simba.bark()
    juno.bark()
    simba.bark()
    print(simba.__dict__)
    print(juno.__dict__)
1. What **variables** does each instance store?
2. What **methods** can you call on an instance?
3. What happens when you make a **new** one?
.\_\_dict\_\_

pedagogical tool
Classes Review

dog.py

class Dog:
    def __init__(self):
        self.times_barked = 0

    def bark(self):
        print('woof')
        self.times_barked += 1

life.py

def main():
    simba = Dog()
    juno = Dog()

    simba.bark()
    juno.bark()
    simba.bark()

    print(simba.__dict__)
    print(juno.__dict__)

Piech + Sahami, CS106A, Stanford University
Did I mention that a class is like a fancy dictionary?
What is self?

When authoring a class, `self` means: "the instance (aka object) I am currently working with"
What does a class do?
A class defines a new variable type
How many variables for the ball?

1. oval
2. change_x
3. change_y
How many variables for 3 balls?
# 1: Store a list of dictionaries

# 2: Store a list of Balls
Recall Functions?

Coder: **Function Author**

- Writes helper functions others can use

Coder: **Function Caller**

- Uses helper functions
Classes also split up the work!

Coder: **Class Author**

- Writes the class (often in its own file), thus defining a new variable type

Coder: **Class Client**

- Uses the new variable type to solve problems (often from main).

Because they are classy.
**Class Client**: Uses the new variable type to solve problems (often from main).

**Class Author**: Writes the class, thus defining a new variable type (often in its own file).
Next step in writing large programs: Better understand memory
You are now ready...
def main():
    x = 5
    print(id(x))
    x += 1
    print(id(x))
What does this do?

```python
def main():
    x = 5
    print(id(x))
    x += 1
    print(id(x))
```

```
stack

main

x

heap

overhead

value

4563589904

5
```
def main():
    x = 5
    print(id(x))
    x += 1
    print(id(x))

What does this do?
What does this do?

def main():
    x = 5
    print(id(x))
    x += 1
    print(id(x))

stack

heap

int

value

1

5

type

ref count

Piech + Sahami, CS106A, Stanford University
def main():
    x = 5
    print(id(x))
    x += 1
    print(id(x))

What does this do?
What does this do?

def main():
    x = 5
    print(id(x))
    x = x + 1
    print(id(x))

stack

heap

main

x

4563589904

4563589904

type

int

value

1

ref count

5
def main():
    x = 5
    print(id(x))
    x = x + 1
    print(id(x))

What does this do?
def main():
    x = 5
    print(id(x))
    x = x + 1
    print(id(x))
What does this do?

```python
def main():
    x = 5
    print(id(x))
    x = x + 1
    print(id(x))
```

![Diagram showing stack and heap for the given code snippet. The stack contains a function `main` with a variable `x` on the heap, which points to an integer value. The heap shows two int nodes: one with a value of 5 and another with a value of 6.](image-url)
The stack

Each time a function is called, a new frame of memory is created.

Each frame has space for all the local variables declared in the function, and parameters.

Each variable has a reference which is like a URL.

When a function returns, its frame is destroyed.
The heap

<table>
<thead>
<tr>
<th>Type</th>
<th>Value</th>
<th>Ref Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>int</td>
<td>1</td>
<td>6</td>
</tr>
</tbody>
</table>

4563589904

Where values are stored

Every value has an address (like a URL address)

Values don’t go away when functions return

Memory is recycled when its no longer used.
def main():
    x = 5
    x = x + 1
What does this do?

```python
def main():
    x = 5
    x = x + 1
```

When a variable is “assigned” via binding you are changing its **reference**

You know a variable is being assigned to if it is on the left hand side of an `=` sign.
def main():
    x = 5
    x = x + 1

What does this do?

When a variable is “used” you are accessing its value
What does this do?

```python
def main():
    x = 5
    binky(9)

def binky(y):
    pinky(y)

def pinky(z):
    print(z)
```

Stack

```
main
x
```
def main():
    x = 5
    binky(9)

def binky(y):
    pinky(y)

def pinky(z):
    print(z)
What does this do?

def main():
    x = 5
    binky(9)

def binky(y):
    pinky(y)

def pinky(z):
    print(z)
def main():
    x = 5
    binky(9)

def binky(y):
    pinky(y)

def pinky(z):
    print(z)

What does this do?
def main():
    x = 5
    binky(9)

def binky(y):
    pinky(y)

def pinky(z):
    print(z)

What does this do?
What does this do?

```python
def main():
    x = 5
    binky(9)

def binky(y):
    pinky(y)

def pinky(z):
    print(z)
```

Stack:
- main
- x: 5563936
- binky
- y: 9563936

Variable Values:
- x: 5
- y: 9

Integers:
- 1
- 5
- 9
def main():
    x = 5
    binky(9)

def binky(y):
    pinky(y)

def pinky(z):
    print(z)
What does this do?

```python
def main():
    x = 5
    binky(9)

def binky(y):
    pinky(y)

def pinky(z):
    print(z)
```

Stack:
- main
  - x: 5563936
  - binky
    - y: 9563936
      - pinky
        - z: 9563936
What does this do?

def main():
    x = 5
    binky(9)

def binky(y):
    pinky(y)

def pinky(z):
    print(z)
```python
def main():
    x = 5
    binky(9)

def binky(y):
    pinky(y)

def pinky(z):
    print(z)
```

**What does this do?**

The code defines three functions: `main()`, `binky()`, and `pinky()`. When `main()` is called, it assigns `x` to 5 and then calls `binky()`. The `binky()` function calls `pinky()` with the argument `y`. The `pinky()` function simply prints its argument `z`.

The stack diagram shows the state of the program as it executes:

- The `main()` function is on the stack with `x` set to 5.
- The `binky()` function is called, with `y` set to 9.
- `pinky()` is called next, with `z` set to 9.

The diagram also illustrates the values on the stack:

- `x`: 5
- `y`: 9
- `z`: 9

The output of running this code would be:

```
5
9
```
def main():
    x = 5
    binky(9)

def binky(y):
    pinky(y)

def pinky(z):
    print(z)
What does this do?

def main():
    x = 5
    binky(9)

def binky(y):
    pinky(y)

def pinky(z):
    print(z)
def main():
    x = 5
    binky(9)

def binky(y):
    pinky(y)

def pinky(z):
    print(z)

Stack:
- main
- x: 5

int:
- 1
- 5
def main():
    x = 5
    binky(9)

def binky(y):
    pinky(y)

def pinky(z):
    print(z)
This is the real matrix…
def main():
    x = ['a', 'b', 'c']
    update(x)

def update(x):
    for v in x:
        print(type(v), v)
        v = v + '!
        print(v)

if __name__ == '__main__':
    main()
What is self?
class Dog:
    def __init__(self, name):
        self.name = new_name
        print(self.name)

# put in another file...
def main():
    first = Dog('simba')

    print(type(first))
    print(id(first))
    print(first.__dict__)
class Dog:
    def __init__(self, new_name):
        self.name = new_name
        print(self.name)

# put in another file...
def main():
    first = Dog('simba')
    second = Dog('juno')

    print(type(first))
    print(id(first))
    print(first.__dict__)
class Dog:
    def __init__(self, new_name):
        self.name = new_name
        print(self.name)

# put in another file...
def main():
    first = Dog('simba')
    second = Dog('juno')
    print(type(first))
    print(id(first))
    print(first.__dict__)

Stack
main
first
second

Heap
Dog
1
reference count
What does this do?

class Dog:

    def __init__(self, new_name):
        self.name = new_name
        print(self.name)

# put in another file...
def main():
    first = Dog('simba')
    second = Dog('juno')
    print(type(first))
    print(id(first))
    print(first.__dict__)

Stack:
- main
- first
- second
- Dog.__init__
- self 42
- new_name 'simba'

Heap:
- Dog
- 1

reference count
What does this do?

class Dog:
    def __init__(self, new_name):
        self.name = new_name
        print(self.name)

# put in another file...
def main():
    first = Dog('simba')
    second = Dog('juno')
    print(type(first))
    print(id(first))
    print(first.__dict__)

Piech + Sahami, CS106A, Stanford University
What does this do?

class Dog:
    def __init__(self, new_name):
        self.name = new_name
        print(self.name)

# put in another file...
def main():
    first = Dog('simba')
    second = Dog('juno')
    print(type(first))
    print(id(first))
    print(first.__dict__)

Stack
- main
- first
- second
- Dog.__init__
  - self
  - new_name: 42
- Dog
  - name: 'simba'

Heap
- Dog
  - reference count: 1
  - name: 'simba'
What does this do?

class Dog:
    def __init__(self, new_name):
        self.name = new_name
        print(self.name)

# put in another file...
def main():
    first = Dog('simba')
    second = Dog('juno')
    print(type(first))
    print(id(first))
    print(first.__dict__)

Stack
main
first
second
Dog.__init__
self 42
new_name 'simba'

Heap
Dog 42
reference count
1
name
'simba'
What does this do?

class Dog:
    def __init__(self, new_name):
        self.name = new_name
        print(self.name)

# put in another file...
def main():
    first = Dog('simba')
    second = Dog('juno')

    print(type(first))
    print(id(first))
    print(first.__dict__)

Stack
    main
    first
    second

Heap
    Dog
        1
        reference count
        name
            ‘simba’

Piech + Sahami, CS106A, Stanford University
```python
class Dog:
    def __init__(self, new_name):
        self.name = new_name
        print(self.name)

# put in another file...
def main():
    first = Dog('simba')
    second = Dog('juno')
    print(type(first))
    print(id(first))
    print(first.__dict__)
```

class Dog:
    def __init__(self, new_name):
        self.name = new_name
        print(self.name)

# put in another file...
def main():
    first = Dog('simba')
    second = Dog('juno')
    print(type(first))
    print(id(first))
    print(first.__dict__)

Stack

main
first 42
second

Heap
Dog
42

reference count
1
name
‘simba’
What does this do?

class Dog:
    def __init__(self, new_name):
        self.name = new_name
        print(self.name)

# put in another file...
def main():
    first = Dog('simba')
    second = Dog('juno')
    print(type(first))
    print(id(first))
    print(first.__dict__)
What does this do?

class Dog:
    def __init__(self, new_name):
        self.name = new_name
        print(self.name)

# put in another file...
def main():
    first = Dog('simba')
    second = Dog('juno')
    print(type(first))
    print(id(first))
    print(first.__dict__)

Stack

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>main</td>
<td></td>
</tr>
<tr>
<td>first</td>
<td>42</td>
</tr>
<tr>
<td>second</td>
<td></td>
</tr>
</tbody>
</table>

Heap

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Reference Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dog</td>
<td>'simba'</td>
<td>1</td>
</tr>
<tr>
<td>Dog</td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>

(reference count)
What does this do?

class Dog:
    def __init__(self, new_name):
        self.name = new_name
        print(self.name)

# put in another file...
def main():
    first = Dog('simba')
    second = Dog('juno')
    print(type(first))
    print(id(first))
    print(first.__dict__)

Stack
main
first 42
second
Dog.__init__
self 48
    new_name 'juno'

Heap
Dog
    reference count
    1
name
    'simba'
    reference count
    48
Dog
    reference count
    1

Piech + Sahami, CS106A, Stanford University
class Dog:
    def __init__(self, new_name):
        self.name = new_name
        print(self.name)

# put in another file...

def main():
    first = Dog('simba')
    second = Dog('juno')

    print(type(first))
    print(id(first))
    print(first.__dict__)

Stack
main
first 42
second
Dog.__init__
self 48
new_name ‘juno’

Heap
Dog
reference count
1
name
‘simba’

Dog
reference count
1
name
‘juno’
What does this do?

class Dog:
    def __init__(self, new_name):
        self.name = new_name
        print(self.name)

# put in another file...

def main():
    first = Dog('simba')
    second = Dog('juno')
    print(type(first))
    print(id(first))
    print(first.__dict__)

Stack

main
first 42
second
Dog.__init__
self 48
new_name 'juno'

Heap

Dog

reference count
1
name
'simba'

Dog

reference count
1
name
'juno'
What does this do?

class Dog:
    def __init__(self, new_name):
        self.name = new_name
        print(self.name)

# put in another file...
def main():
    first = Dog('simba')
    second = Dog('juno')
    print(type(first))
    print(id(first))
    print(first.__dict__)

Stack
main
first 42
second
Dog.__init__
self 48
new_name 'juno'

Heap
Dog
reference count
1
name
'simba'

Dog
reference count
1
name
'juno'
def main():
    first = Dog('simba')  
    second = Dog('juno')

    print(type(first))
    print(id(first))
    print(first.__dict__)

Stack
main
first 42
second

Heap
Dog
  name
    ‘simba’
  reference count
    1

Dog
  name
    ‘juno’
  reference count
    1

print(first.name)

# put in another file...

class Dog:
    def __init__(self, new_name):
        self.name = new_name
        print(self.name)
What does this do?

```python
class Dog:
    def __init__(self, new_name):
        self.name = new_name
        print(self.name)

# put in another file...
def main():
    first = Dog('simba')
    second = Dog('juno')
    print(type(first))
    print(id(first))
    print(first.__dict__)
```

Stack:
- main
- first 42
- second 48

Heap:
- Dog
  - reference count: 1
  - name: 'simba'
- Dog
  - reference count: 1
  - name: 'juno'

Piech + Sahami, CS106A, Stanford University
class Dog:
    def __init__(self, new_name):
        self.name = new_name
        print(self.name)

# put in another file...
def main():
    first = Dog('simba')
    second = Dog('juno')
    print(type(first))
    print(id(first))
    print(first.__dict__)
What does this do?

class Dog:
    def __init__(self, new_name):
        self.name = new_name
        print(self.name)

# put in another file...
def main():
    first = Dog('simba')
    second = Dog('juno')

    print(type(first))
    print(id(first))
    print(first.__dict__)
What does this do?

class Dog:
    def __init__(self, new_name):
        self.name = new_name
        print(self.name)

# put in another file...

def main():
    first = Dog('simba')
    second = Dog('juno')
    print(type(first))
    print(id(first))
    print(first.__dict__)

Stack

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>main</td>
<td></td>
</tr>
<tr>
<td>first</td>
<td>42</td>
</tr>
<tr>
<td>second</td>
<td>48</td>
</tr>
</tbody>
</table>

Heap

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>42</td>
<td></td>
</tr>
<tr>
<td>Dog</td>
<td></td>
</tr>
</tbody>
</table>

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>name</td>
<td></td>
</tr>
<tr>
<td>'simba'</td>
<td></td>
</tr>
</tbody>
</table>

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>48</td>
<td></td>
</tr>
<tr>
<td>Dog</td>
<td></td>
</tr>
</tbody>
</table>

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>name</td>
<td></td>
</tr>
<tr>
<td>'juno'</td>
<td></td>
</tr>
</tbody>
</table>
Challenge: Trace This!

dog.py

class Dog:
    def __init__(self):
        self.times_barked = 0

    def bark(self):
        print('woof')
        self.times_barked += 1

life.py

def main():
    simba = Dog()
    juno = Dog()
    simba.bark()
    juno.bark()
    simba.bark()
    print(simba.__dict__)
    print(juno.__dict__)

Piech + Sahami, CS106A, Stanford University
1. Practice with classes
2. See how to trace memory with classes
Guiding question for today:
what does it take to go from what you know to writing big-scale software?
Bouncing Balls
What does a class do?
A class defines a new variable type