

objects

objectify Python, not people



important primer

<https://www.youtube.com/watch?v=jalHcE7gZn8#t=17>

objects

- used to model real-world things
- have attributes (think nouns)
- have methods (think verbs)

how to

1. make a class
2. make an instance of the class
3. use the object

attributes

- create/access them through special **self** argument
- only create them within **__init__** method (best practice)

attributes

```
class Sloth(object):  
  
    def __init__(self, name, stuffie):  
        self.name = name  
        self.stuffie = stuffie  
  
s = Sloth('Punkin', 'Giraffe')  
  
# Get attributes  
print s.stuffie  
  
# Change attributes  
s.stuffie = 'Manatee'  
print s.stuffie
```

methods

```
class Sloth(object):  
    # initializer, etc.  
  
    def hug(self):  
        return '{0} hugs {1}'.format(self.name,  
                                     self.stuffie)  
  
s = Sloth('Josh', 'Sloth')  
  
# a kid can dream  
print s.hug()
```


more methods

```
class Sloth(object):  
    # initializer, etc.  
  
    def change_stuffie(self, new_stuffie):  
        if new_stuffie is None:  
            raise ValueError('Give that sloth a stuffie')  
  
        self.stuffie = new_stuffie  
  
s = Sloth('Punkin', 'Giraffe')  
  
s.change_stuffie(None) # Raises an error  
s.change_stuffie('Manatee')
```

full sloth class

```
class Sloth(object):  
    def __init__(self, name, stuffie):  
        self.name = name  
        self.stuffie = stuffie  
  
    def hug(self):  
        return '{0} hugs {1}'.format(self.name,  
                                     self.stuffie)  
  
    def change_stuffie(self, new_stuffie):  
        if new_stuffie is None:  
            raise ValueError('Give that sloth a stuffie')  
  
        self.stuffie = new_stuffie
```

class attributes

- single attribute available to entire class
- access two ways:
 - `Class.attr`
 - `instance.attr`

class attribute example

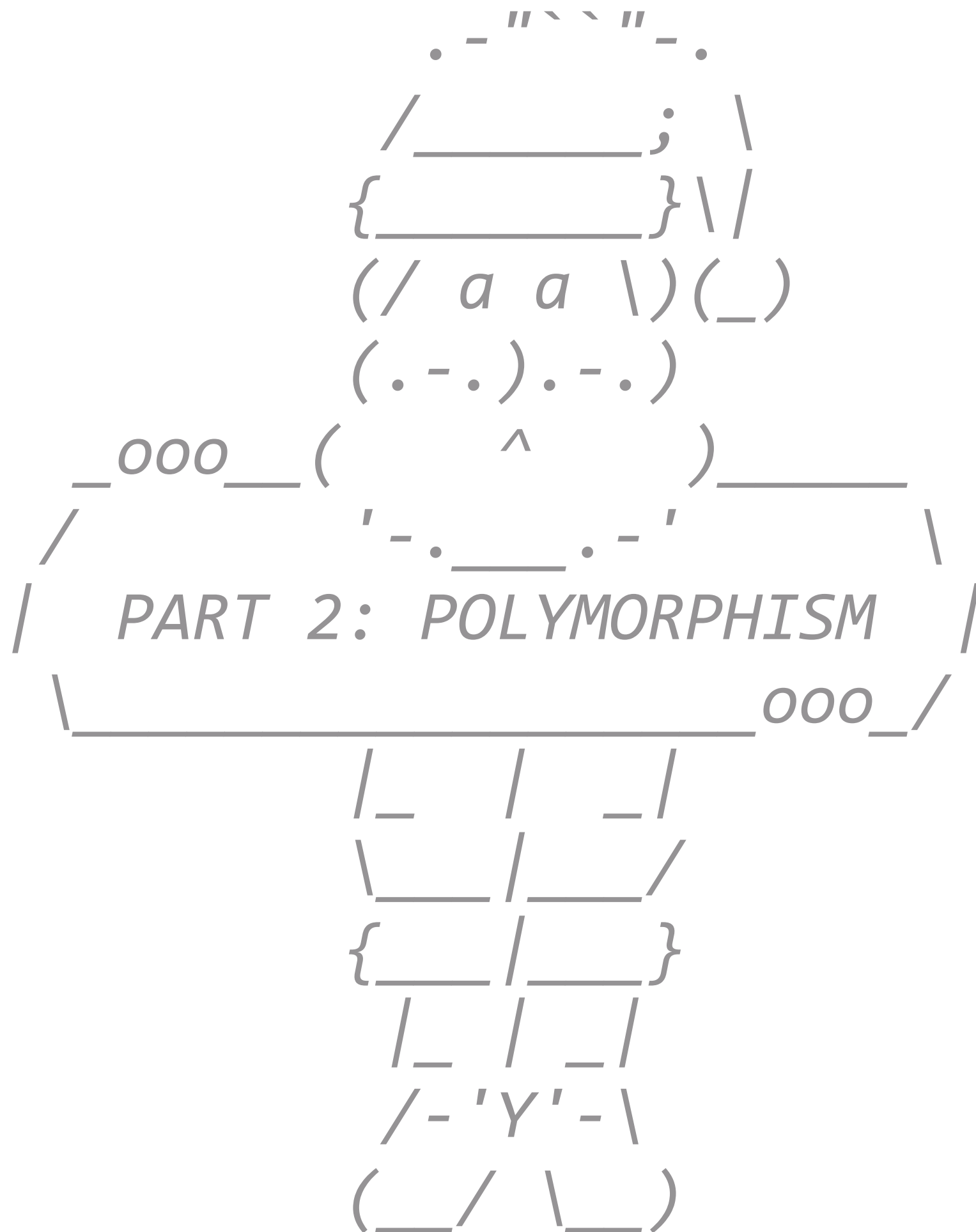
```
class Person(object):  
    all = []  
  
    def __init__(self, name):  
        self.name = name  
        self.all.append(self)
```

```
p = Person("Josh Palay")  
print p  
# <__main__.Person object at 0x10e112250>
```

```
Person.all  
# [<__main__.Person object at 0x10e112250>]
```

```
p.all  
# [<__main__.Person object at 0x10e112250>]
```

```
class Node(object):  
    def __init__(self, val, tail):  
        self.val = val  
        self.tail = tail  
  
    def has_cycle(self):  
        nodes = set()  
        curr = self  
  
        while curr is not None:  
            if curr in nodes:  
                return True  
  
            nodes.add(curr)  
            curr = curr.tail  
  
        return False
```



classic example

- Let's model...
 - People!
 - Employees!
 - Students!
- Students/Employees have everything People have

without polymorphism

```
class Person(object):  
    # person attributes + methods  
  
class Employee(object):  
    # person attributes + methods  
    # employee attributes + methods  
  
class Student(object):  
    # person attributes + methods  
    # student attributes + methods
```


with polymorphism

```
class People(object):  
    # people attrs/methods
```

```
class Employee(People):  
    # employee attrs/methods
```

```
class Student(People):  
    # student attrs/methods
```

side-by-side

without polymorphism:

```
class Person(object):  
    # person attrs/methods
```

```
class Employee(object):  
    # person attrs/methods  
    # employee attrs/methods
```

```
class Student(object):  
    # person attrs/methods  
    # student attrs/methods
```

with polymorphism:

```
class People(object):  
    # people attrs/methods
```

```
class Employee(People):  
    # employee attrs/methods
```

```
class Student(People):  
    # student attrs/methods
```

classic examples are boring

```
class Storm(object):  
  
    def __init__(self, windspeed):  
        self.wind = windspeed  
  
    def get_wind(self):  
        return self.wind  
  
    def get_report(self):  
        return 'Winds are blowing at {0}mph!'.format(  
            self.wind)
```

other types of storms

- Consider the [Sharknado](#)
- Employee : Person :: Sharknado : Storm
- Sharknado should extend Storm

initializer

```
class Sharknado(Storm):  
  
    def __init__(self, windspeed, shark_level):  
        # calls Storm's initializer  
        super(Sharknado, self).__init__(windspeed)  
  
        self.shark_level = shark_level
```

get_report

```
class Sharknado(Storm):  
  
    def get_report(self):  
        # Calls Storm's get_report  
        r = super(Sharknado, self).get_report()  
  
        r += 'Also {0} sharks bit me :('.format(  
            self.shark_level)  
        return r
```

get_wind

/\ /\
 /`_ , --="=-- ,_//`/
 \ . " : ' . . ' : " . /
) : ' : (
 />/0\ /0\
 / \ - " ~ \ ~ " - / /
 > / ` === . _ / . === ` / <
 . - " - . \ === ' / ' === / . - " - .
 . --- { ' . ' ` } --- \ , . - ' - . , / --- { . ' . ' } --- .
) ` " _ _ _ " ` ~ - = = = - ~ ` " _ _ _ " ` (
 (Part 3 (bonus! :o):)
) Other things (
 ' _____ '

range is a thing

- built in function
- makes lists of numbers

range is a thing

creates list [0, end)
`range(end)`

creates list [start, end)
`range(start, end)`

if step = n, creates list with every nth number
`range(start, end, step)`

range is a thing

```
# creates list [0...9]  
range(10)
```

```
# creates list [1...10]  
range(1, 11)
```

```
# creates list [0, 3, 6, 9]  
range(0, 10, 3)
```

xrange is better

- Returns "generator", not a list
- Uses less memory
- Usually does what you want

list comprehensions

- easy way to transform list
- returns new list, never modifies old one
- [`<statement> for elt in lst`]

list comprehensions

```
nums = xrange(5)
squared_nums = [x ** 2 for x in nums]

print squared_nums
# [0, 1, 4, 9, 16]
```

you can also filter

```
nums = xrange(10)
odds = [x for x in nums if x % 2 == 1]

print odds
# [1, 3, 5, 7, 9]
```

do both!

```
nums = xrange(10)
odds_sq = [x ** 2 for x in nums if x % 2 == 1]

print odds_sq
# [1, 9, 25, 49, 81]
```