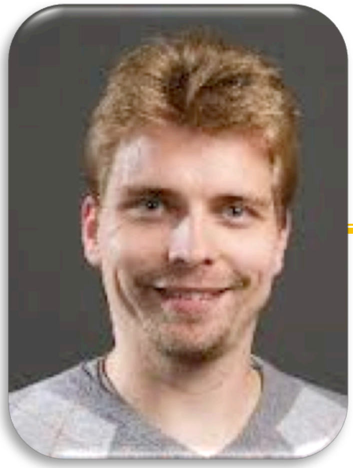


Computational Structures in Data Science

UC Berkeley EECS
Adj. Ass. Prof.
Dr. Gerald Friedland

Lecture #09: Object-Oriented Programming

April Fool’s Day, 2019

http://inst.eecs.berkeley.edu/~cs88

Administrivia

- Welcome back from Spring Break!
- Class becomes a lot more practical from here on.
- Beware of April fools day!

Solutions for the Wandering Mind

Consider the following Python3 code:

```
_='_=%r;print _(%%)_';print _(%)_
```

What does it do?

It prints itself out! This is called a “quine”.

Can you find other ways to do the same?

Yes, for example:

```
print((lambda s:s%s)('print((lambda s:s%%s)(%r))'))
```

The general idea of a quine is: The source code contains a string of itself, which is output twice, once inside quotation marks.

We need two similar copies of the same to self-replicate, just like DNA!

Computational Concepts Toolbox

- Data type: values, literals, operations,
- Expressions, Call expression
- Variables
- Assignment Statement
- Sequences: tuple, list
- Dictionaries
- Data structures
- Tuple assignment
- Function Definition Statement

- Conditional Statement
- Iteration: list comp, for, while
- Lambda function expr.

- Higher Order Functions
  - Functions as Values
  - Functions with functions as argument
  - Assignment of function values
- Higher order function patterns
  - Map, Filter, Reduce
- Function factories – create and return functions
- Recursion
  - Linear, Tail, Tree
- Abstract Data Types
- Generators
- Mutation
- Object Orientation

Mind Refresher 1

- **A mutation is…**

  A) A monster from a movie
  B) A change of state
  C) Undesirable
  D) All of the above

Solution:
B) A change of state

Mind Refresher 2

- **We try to hide states because…**

A) We don't like them
B) Math doesn't have them
C) It's easier to program not having to think about them
D) All of the above

Solution:

C) It's easier not to have to think about them. Remember: *n* Boolean variables: 2^n states!

Mind Refresher 3

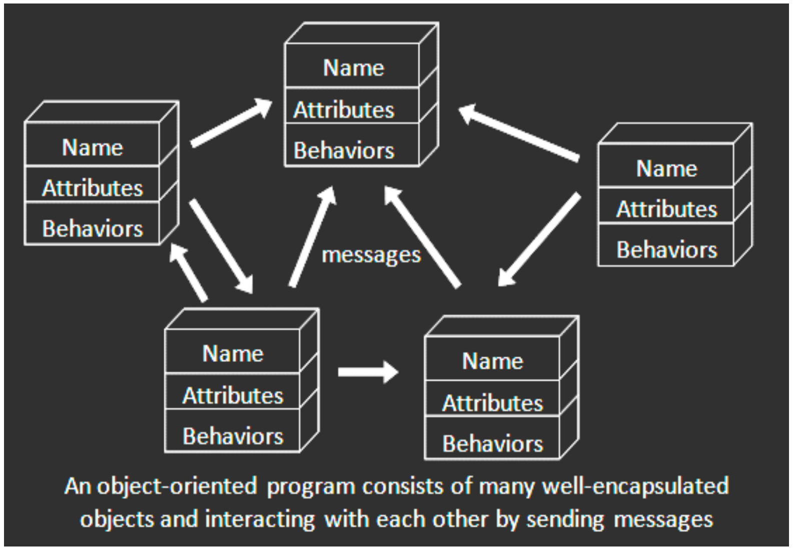
- Where do we hide states?

A) Local variables in functions
B) Private variables in objects
C) Function calls in recursions
D) All of the above

Solution:
D) All of the above

Object-Oriented Programming (OOP)

- Objects as data structures
  - With methods you ask of them
    - » These are the behaviors
  - With local state, to remember
    - » These are the attributes
- Classes & Instances
  - Instance an example of class
  - E.g., Fluffy is instance of Dog
- Inheritance saves code
  - Hierarchical classes
  - E.g., pianist special case of musician, a special case of performer
- Examples (tho not pure)
  - Java, C++

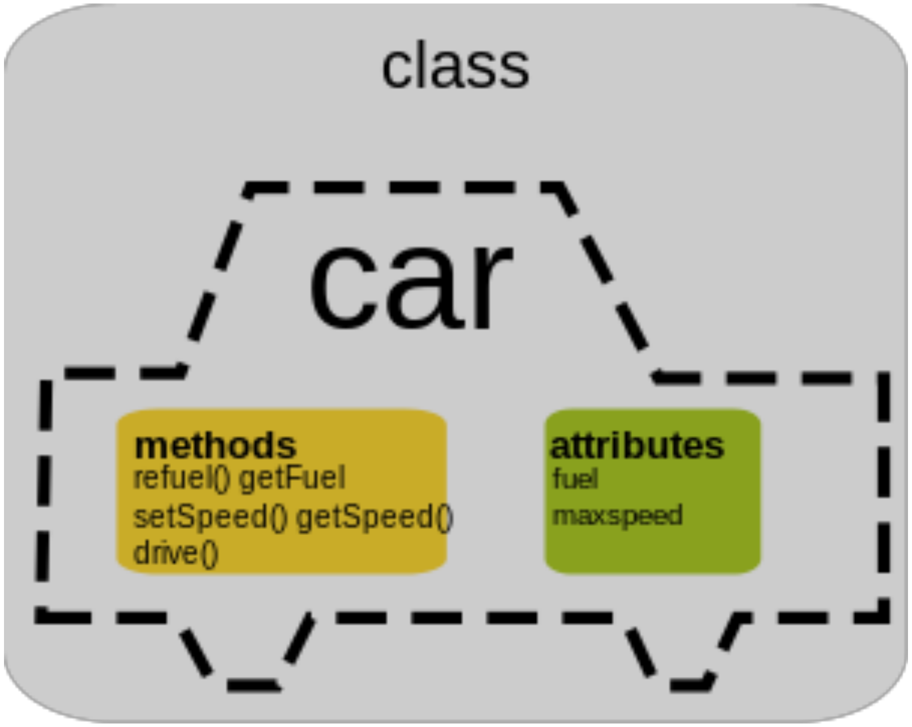

Name
Attributes
Behaviors
messages
An object-oriented program consists of many well-encapsulated objects and interacting with each other by sending messages

`www3.ntu.edu.sg/home/ehchua/programming/java/images/OOP-Objects.gif`

Classes

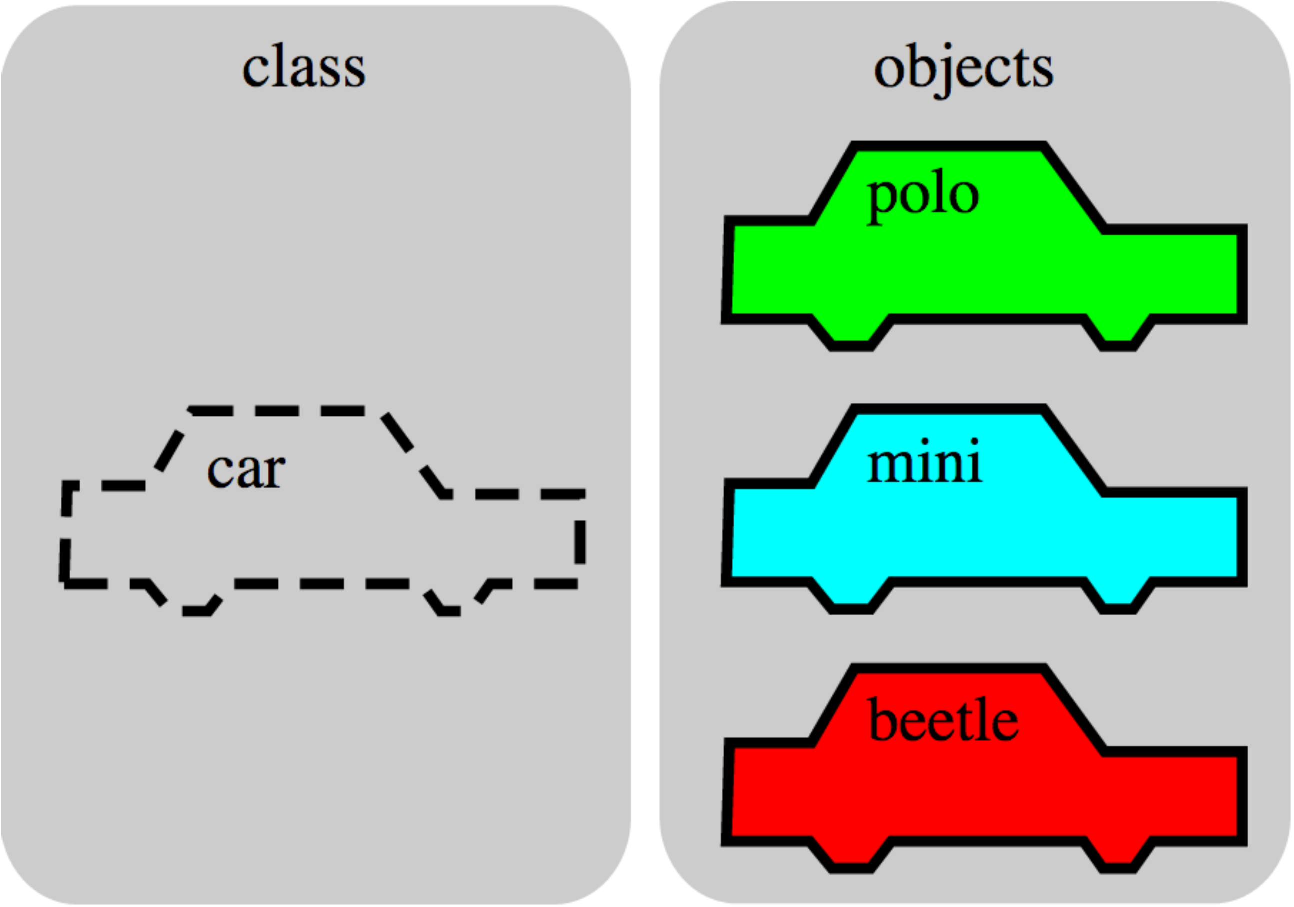
- Consist of data and behavior, bundled together to create abstractions
  - Abstract Data Types
- A class has
  - attributes (variables)
  - methods (functions)

  that define its behavior.



Objects

- An object is the instance of a class.



Objects

- Objects are concrete instances of classes in memory.

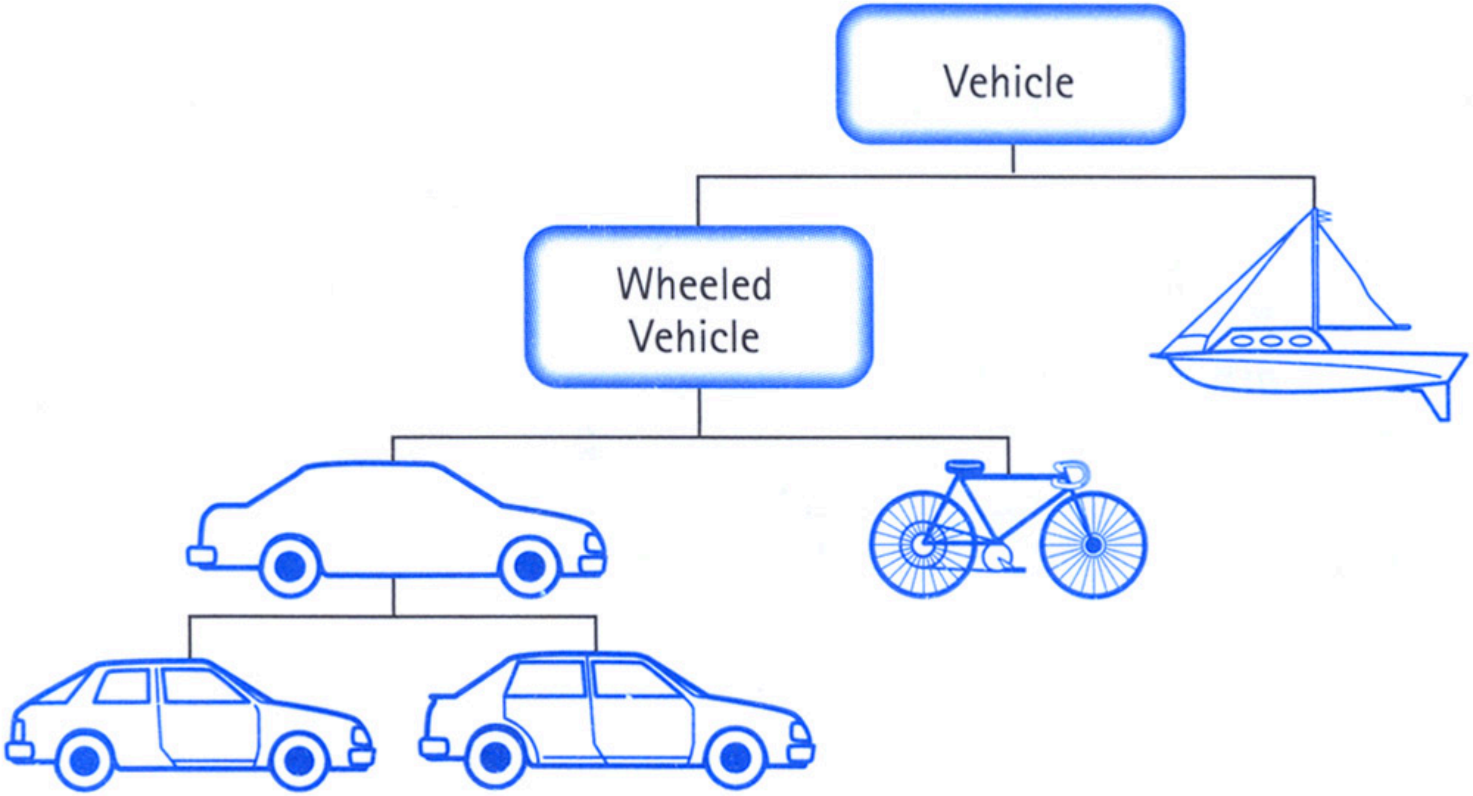
- They can have state
  - mutable vs immutable

- Functions do one thing (well)
  - Objects do a collection of related things

- In Python, everything is an object
  - All objects have attributes
  - Manipulation happens through methods

Class Inheritance

- Classes can inherit methods and attributes from parent classes but extend into their own class.



Inheritance

- Define a class as a specialization of an existing class
- Inherent its attributes, methods (behaviors)
- Add additional ones
- Redefine (specialize) existing ones
  - Ones in superclass still accessible in its namespace

Review: Bank account using dictionary

```
account_number_seed = 1000

def account(name, initial_deposit):
    global account_number_seed
    account_number_seed += 1
    return {'Name' : name, 'Number': account_number_seed,
              'Balance' : initial_deposit}

def account_name(acct):
    return acct['Name']


def account_balance(acct):
    return acct['Balance']

def account_number(acct):
    return acct['Number']

def deposit(acct, amount):
    acct['Balance'] += amount
    return acct['Balance']

def withdraw(acct, amount):
    acct['Balance'] -= amount
    return acct['Balance']
```

```
>>> my_acct = account('David Culler', 100)
>>> my_acct
{'Name': ’John Doe', 'Balance': 100,
'Number': 1001}
>>> account_number(my_acct)
1001
>>> your_acct = account("Fred Jones", 475)
>>> account_number(your_acct)
1002
>>>
```

Python class statement

```
class ClassName:
    <statement-1>
    .
    .
    .
    <statement-N>
```

```
class ClassName ( inherits ):
    <statement-1>
    .
    .
    .
    <statement-N>
```

Example: Account

```
class BaseAccount:

    def init(self, name, initial_deposit):
        self.name = name
        self.balance = initial_deposit

    def account_name(self):

        return self . name

    def account_balance(self):
        return self.balance

    def withdraw(self, amount):
        self.balance -= amount
        return self.balance
```

new namespace

attributes

The object

da dot

methods

Creating an object, invoking a method

The Class Constructor

```
my_acct = BaseAccount()
my_acct.init(”John Doe", 93)
my_acct.withdraw(42)
```

da dot

Special Initialization Method

```
class BaseAccount:

    def __init__(self, name, initial_deposit):
        self.name = name
        self.balance = initial_deposit

    def account_name(self):

        return self.name

    def account_balance(self):
        return self.balance

    def withdraw(self, amount):
        self.balance -= amount
        return self.balance
```

return None

More on Attributes

- Attributes of an object accessible with ‘dot’ notation

```
    obj.attr
```

- Most OO languages provide *private* instance fields for access only inside object
  - Python leaves it to convention

- Class variables vs Instance variables:
  - Class variable set for all instances at once
  - Instance variables per instance value

Example

```
class BaseAccount:

    def __init__(self, name, initial_deposit):
        self.name = name
        self.balance = initial_deposit

    def name(self):
        return self.name

    def balance(self):
        return self.balance

    def withdraw(self, amount):
        self.balance -= amount
        return self.balance
```

Example: “private” attributes

```
class BaseAccount:

    def __init__(self, name, initial_deposit):
        self._name = name
        self._balance = initial_deposit

    def name(self):
        return self._name

    def balance(self):
        return self._balance

    def withdraw(self, amount):
        self._balance -= amount
        return self._balance
```

Example: class attribute

```
class BaseAccount:
    account_number_seed = 1000

    def __init__(self, name, initial_deposit):
        self._name = name
        self._balance = initial_deposit
        self._acct_no = BaseAccount.account_number_seed
        BaseAccount.account_number_seed += 1
    def name(self):
        return self._name

    def balance(self):
        return self._balance

    def withdraw(self, amount):
        self._balance -= amount
        return self._balance
```

More class attributes

```
class BaseAccount:
    account_number_seed = 1000
    accounts = []
    def __init__(self, name, initial_deposit):
        self._name = name
        self._balance = initial_deposit
        self._acct_no = BaseAccount.account_number_seed
        BaseAccount.account_number_seed += 1
        BaseAccount.accounts.append(self)

    def name(self):
        ...

    def show_accounts():
        for account in BaseAccount.accounts:
            print(account.name(),
                  account.account_no(),account.balance())
```

Example

```
class Account(BaseAccount):
    def deposit(self, amount):
        self._balance += amount
        return self._balance
```

More special methods

```
class Account(BaseAccount):
    def deposit(self, amount):
        self._balance += amount
        return self._balance

    def __repr__(self):
        return '< ' + str(self._acct_no) +
               '[' + str(self._name) + '] >'

    def __str__(self):
        return 'Account: ' + str(self._acct_no) +
               '[' + str(self._name) + ']'

    def show_accounts():
        for account in BaseAccount.accounts:
            print(account)
```

Goal: unambiguous

Goal: readable

Classes using classes

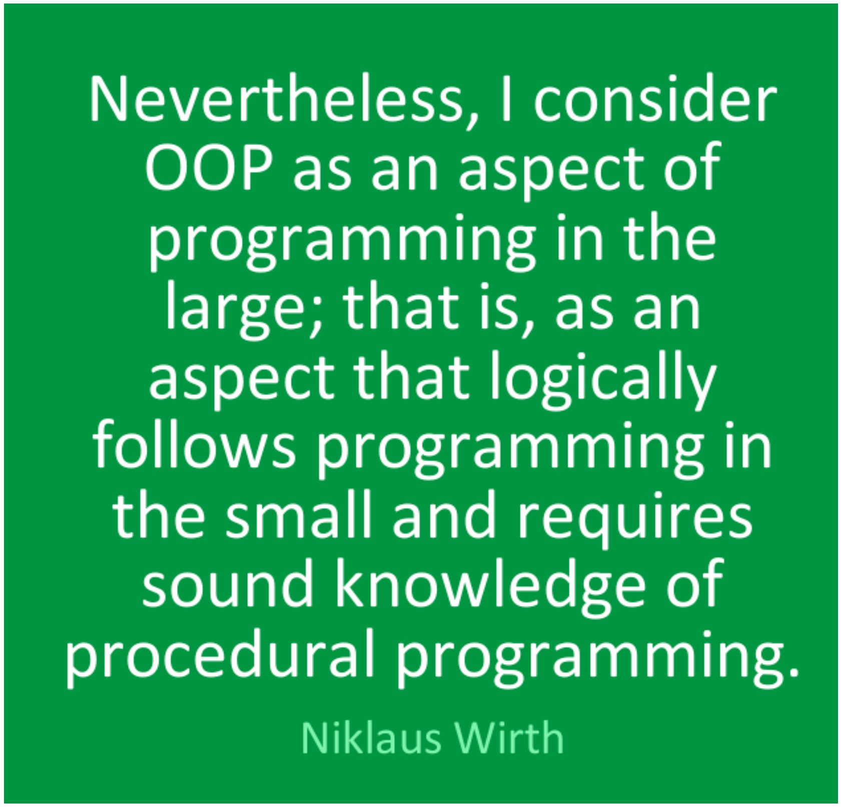
```
class Bank:
    accounts = []

    def add_account(self, name, account_type,
                    initial_deposit):
        assert (account_type == 'savings') or
               (account_type == 'checking'), "Bad Account type"
        assert initial_deposit > 0, "Bad deposit"
        new_account = Account(name, account_type,
                              initial_deposit)
        Bank.accounts.append(new_account)

    def show_accounts(self):
        for account in Bank.accounts:
            print(account)
```

Key concepts to take forward

- Class definition
- Class namespace
- Methods
- Instance attributes (fields)
- Class attributes
- Inheritance
- Superclass reference



Nevertheless, I consider OOP as an aspect of programming in the large; that is, as an aspect that logically follows programming in the small and requires sound knowledge of procedural programming.

Niklaus Wirth

Thoughts for the Wandering Mind

Can you write a quine that mutates on self-replication?

Give an example.