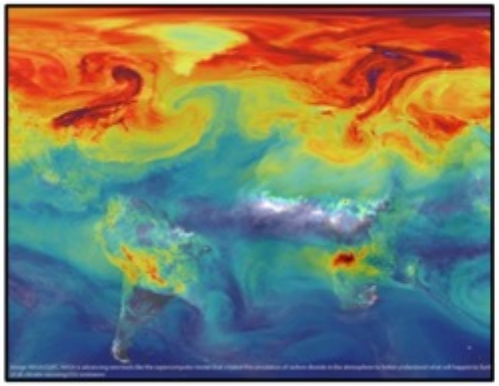





Geo Data Science with Python (GEOS-5984/4984)

Prof. Susanna Werth

Topic: Python Object Oriented Programming & Classes

Today's music is from: Nazmul

Please keep sending me your song suggestions through Canvas!

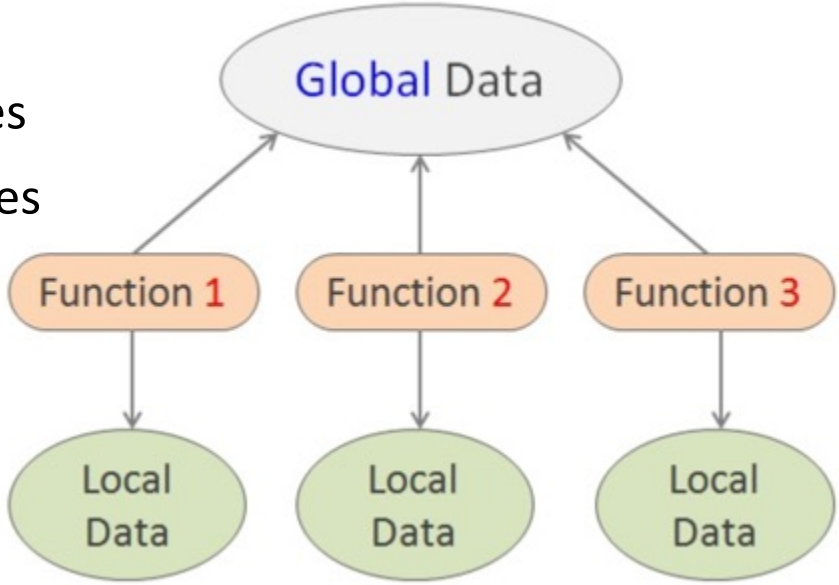
Object Oriented Programming in Python

(OOP)

Programming Paradigms

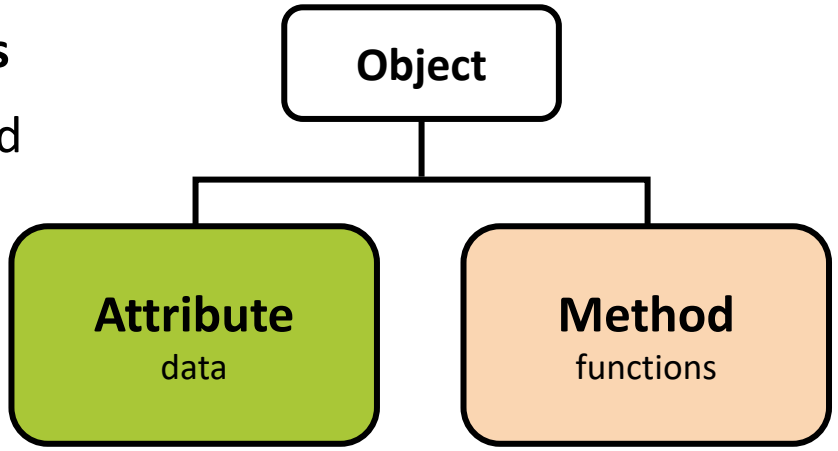
Procedural

- Divides large program into smaller procedures
- Focused on accurate description of procedures (sub-algorithms), e.g., functions
- Functions operate on data (= variables)
- “Tactical mode” (faster programming)
- Examples: C, Fortran, Pascal



Objected-oriented

- Focuses on data, objects and their **properties**
- Objects contain data in form of **attributes** and code in the form of **methods**
- Inheritance hierarchy (btw. object-types)
- “Strategic mode” (efficient programming)
- Examples: C++, Java, Perl, Python
- Python: full OOP ability, but optional



Object-oriented Programming (OOP)

Wikipedia:

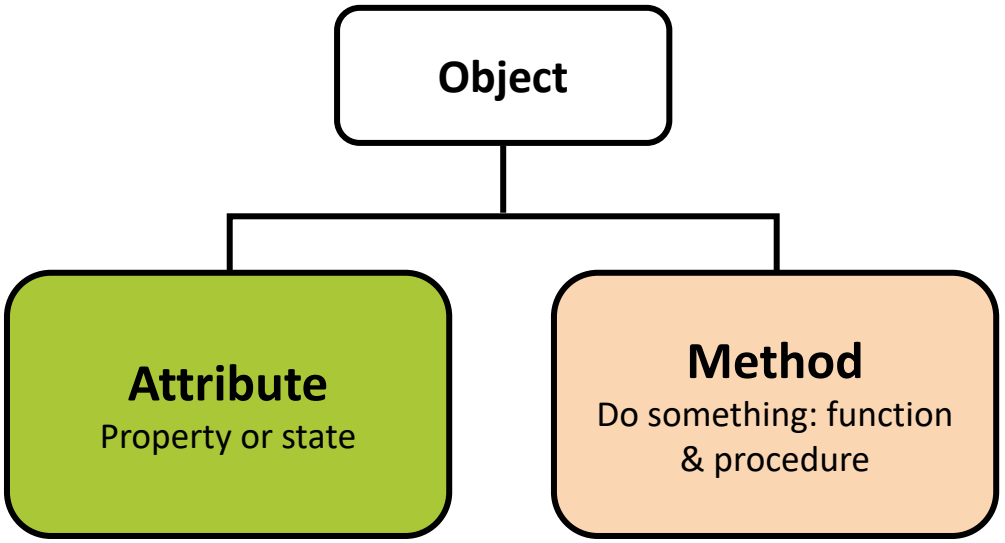
Object-oriented programming (OOP) *is a programming paradigm based on the concept of "**objects**", which may contain data, in the form of (…) **attributes**; and code, in the form of (…) **methods**.*

*A feature of objects is that an object's [**methods**] **can access and often modify the data fields** of the object with which they are associated.*

…

*There is significant diversity of OOP languages, but the most popular ones are **class-based**, meaning that **objects are instances of classes**, which typically also determine their type.*

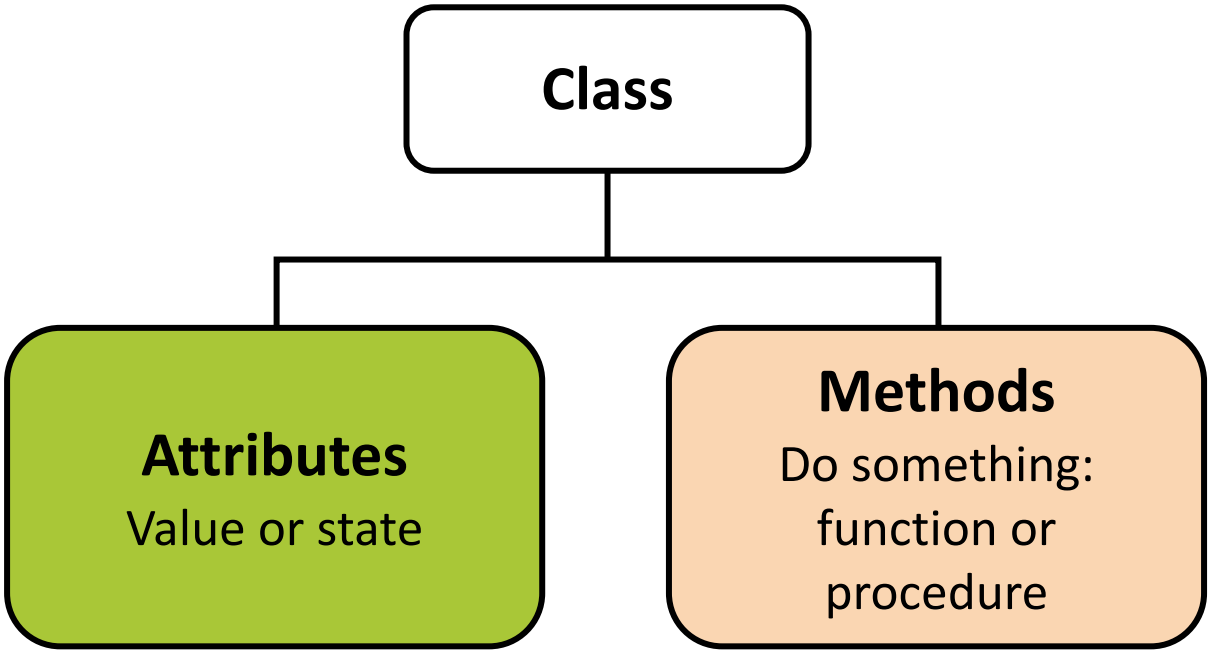
OOP Concept



- Object-based
- Objects have attributes (content/data/propertues) and methods (procedures/functions/tools)
- Objects are instances of classes

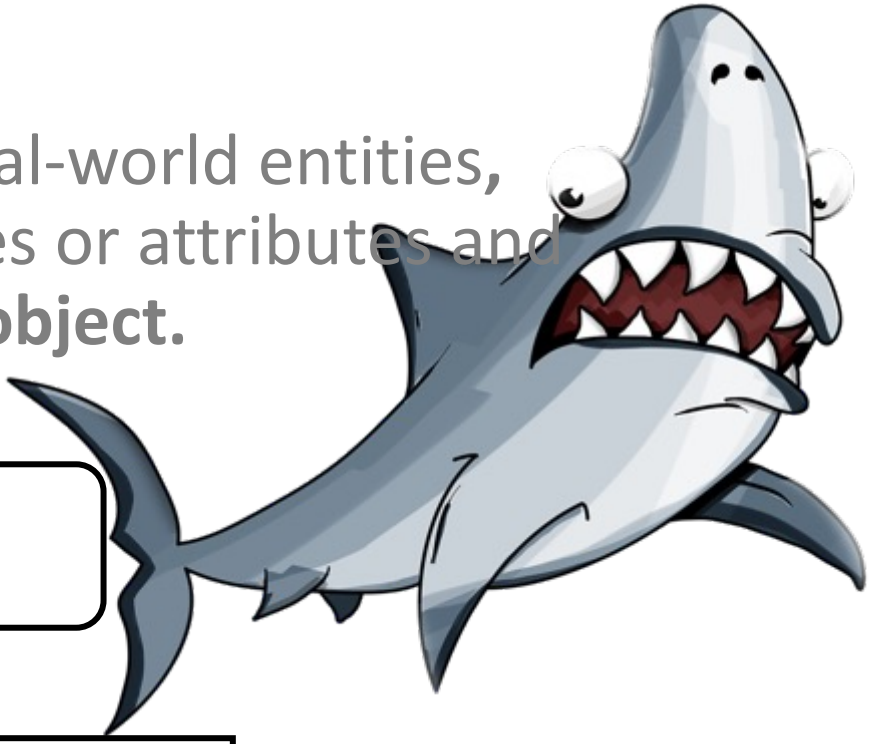
Classes

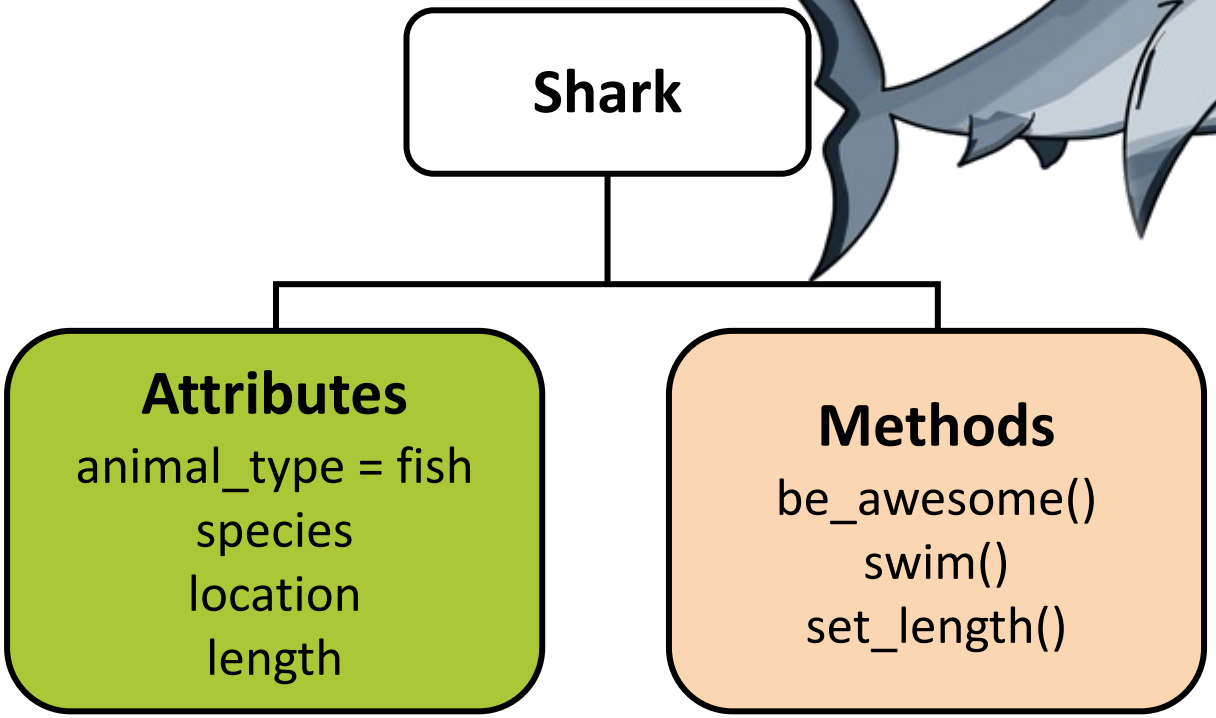
A Class is an abstraction of real-world entities**,** which share generic properties or attributes and behaviors. **A blueprint of an object.**



Classes

A Class is an abstraction of real-world entities, which share generic properties or attributes and behaviors. **A blueprint of an object.**

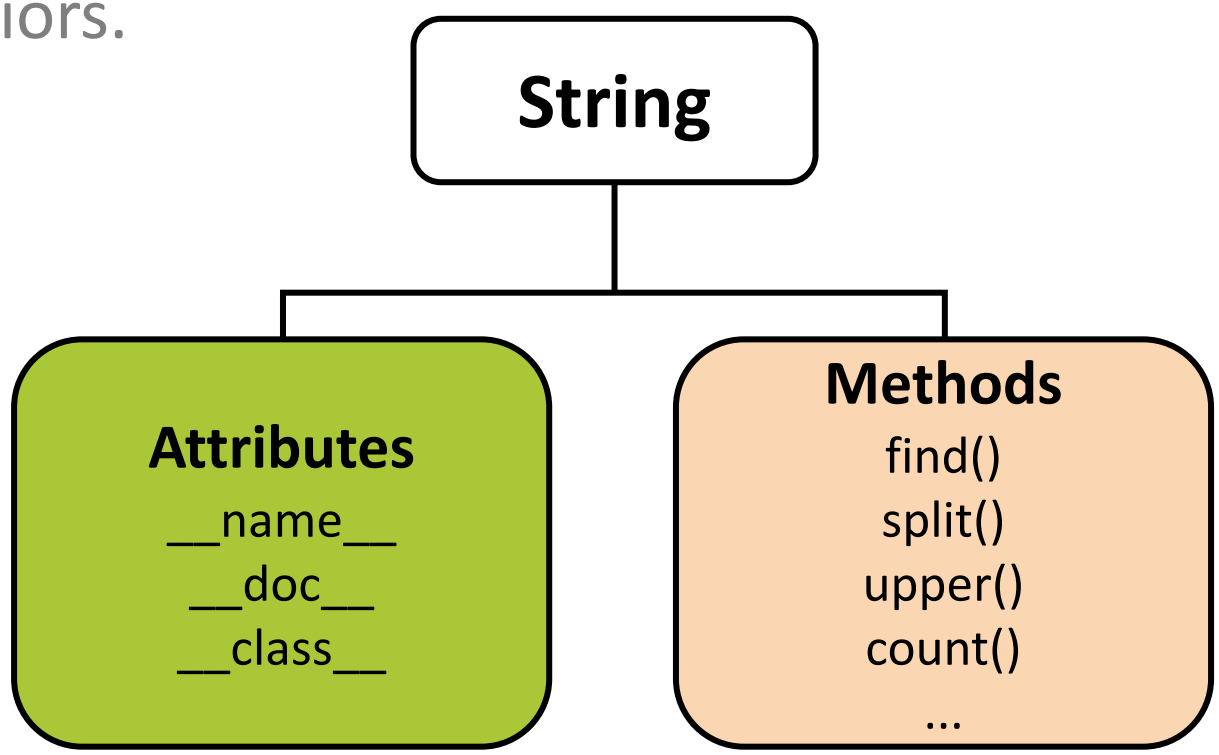




Classes

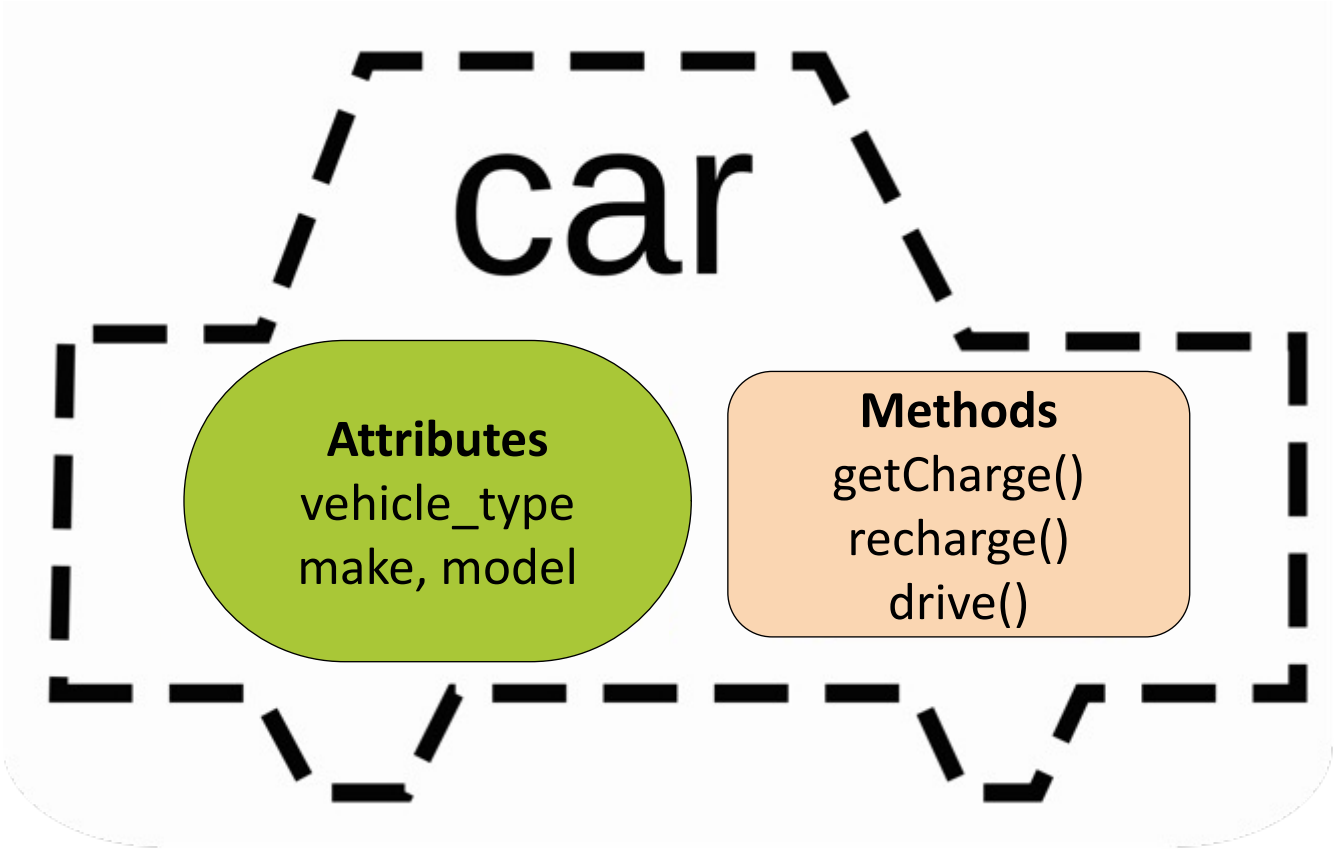
"Hello, world!"

A Class is an abstraction of real-world entities, which share generic properties or attributes and behaviors.

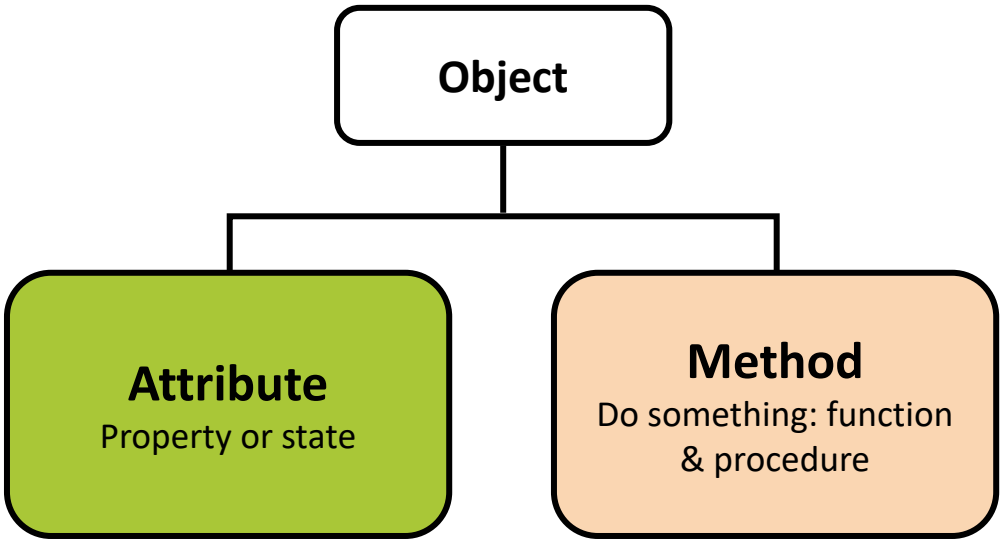


Classes

A Class is an abstraction of real-world entities**,** which share generic properties or attributes and behaviors. **A blueprint of an object.**



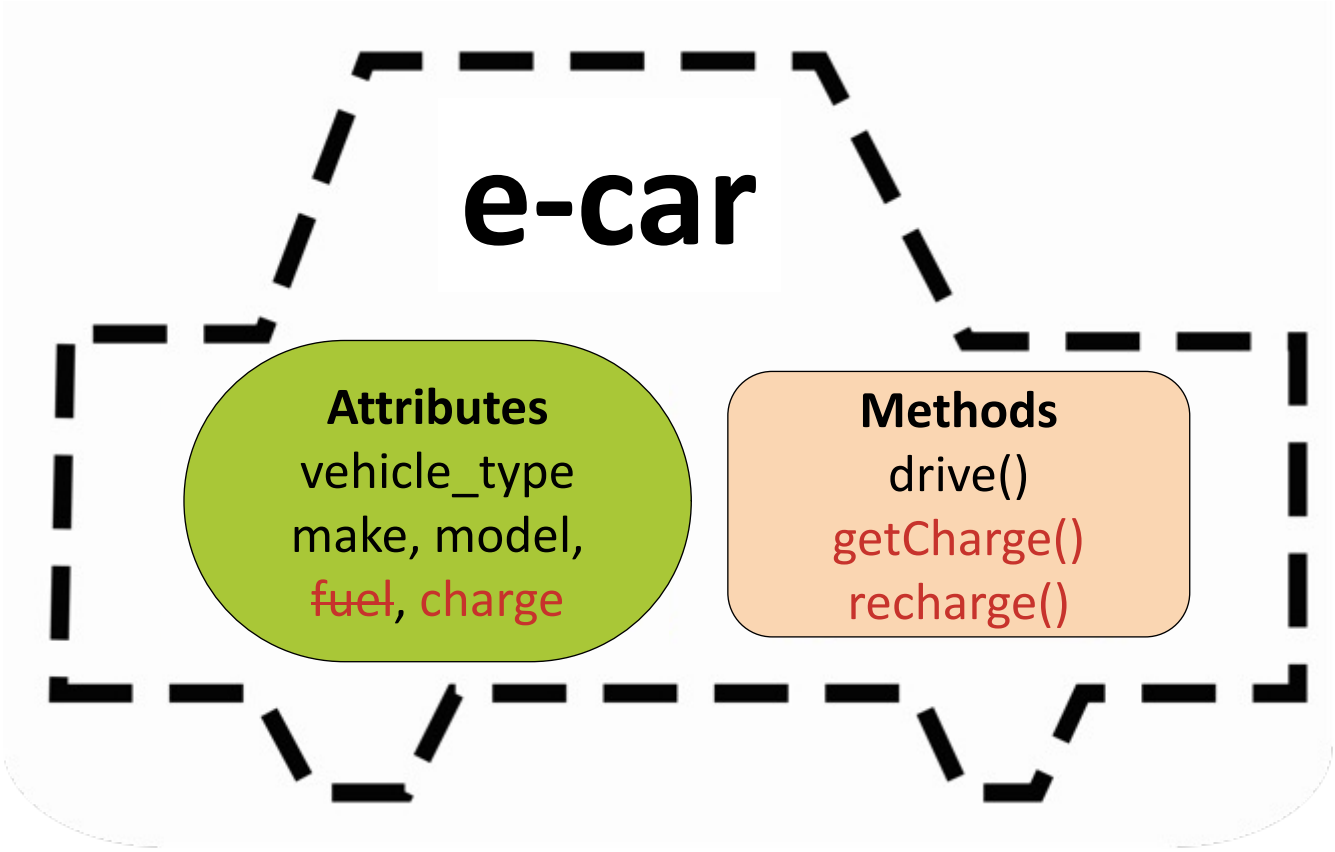
OOP Purpose



- Writing new programs by **customizing existing code** instead of changing it in-place
- Minimize code redundancy
- Different and often more effective way of programming

Classes & Subclasses

Customizing existing code, e.g. Car class

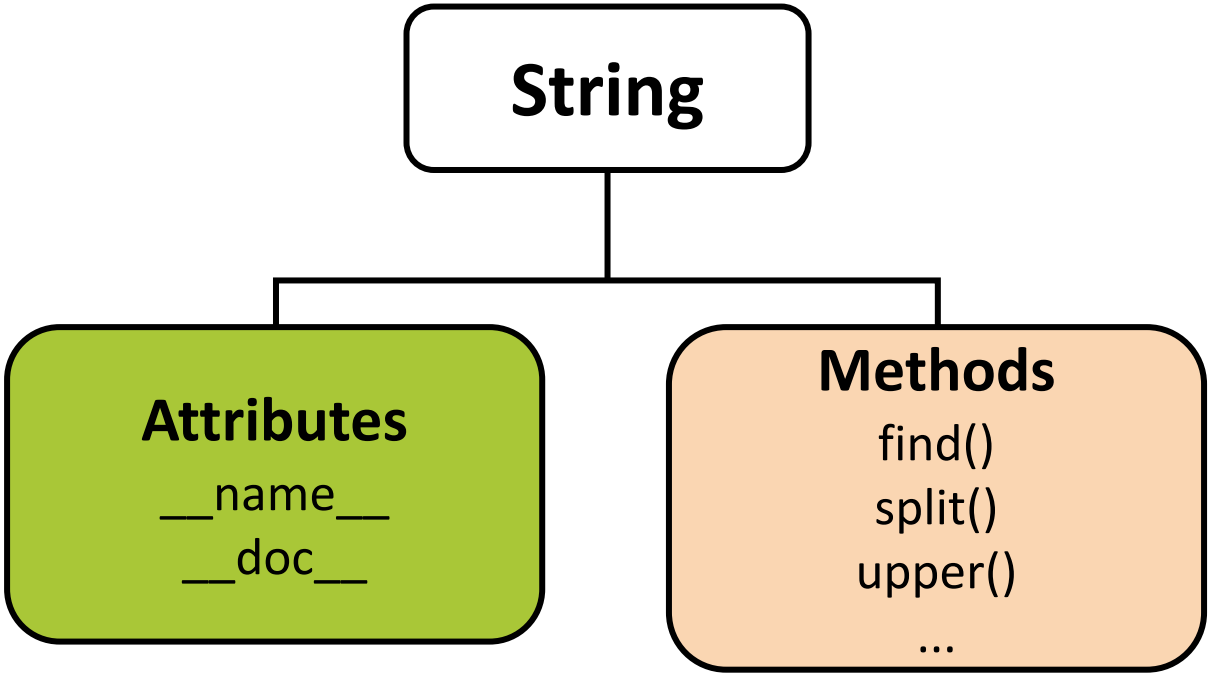


Classes & Subclasses

"Hello, world!"

Customizing existing code

String is a sub-class of the class *type*



Classes in Python

Object-oriented Programming

Core Object Types

Table 4-1. Built-in objects preview

Object type	Example literals/creation
Numbers	`1234, 3.1415, 3+4j, Decimal, Fraction`
Strings	`'spam', "guido's", b'a\x01c'`
Lists	`[1, [2, 'three'], 4]`
Dictionaries	`{'food': 'spam', 'taste': 'yum'}`
Tuples	`(1, 'spam', 4, 'U')`
Files	`myfile = open('eggs', 'r')`
Sets	`set('abc'), {'a', 'b', 'c'}`
Other core types	Booleans, types, `None`
Program unit types	Functions, modules, classes
Implementation-related types	Compiled code, stack tracebacks

Lutz, M. (2013). *Learning Python* (5th ed.). O'Reilly Media, Inc.

Classes in Python

- Python's main OOP tool
- Python program units, with their own namespaces
- Extend idea of modular programming:
  Packages of functions that use and process built-in object types

- **Object factories**: Create and manage new objects
- Useful for any application decomposable into set of objects
- All Python object types are managed by respective classes

Classes & OOP concepts

Compositions

- Objects are composed of other objects
- Each component might be coded as class, defining it's own behavior and relationships

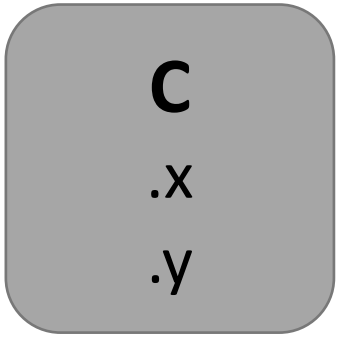
Inheritance

- Objects inherit properties from general category of objects of the same type
- Common properties have to be implemented only once
- Minimize code redundancy, maximize code reuse

Instances of Classes

Class Objects (C)

- Instance factories
- Composed of objects
- Provide behavior that is inherited by all instances
- *Blueprint of an Object*

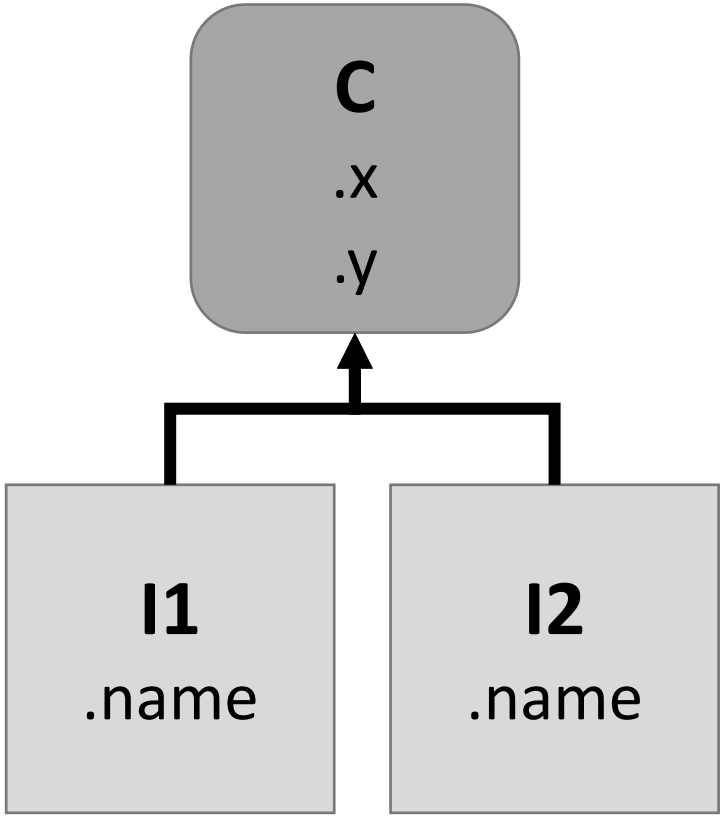


Lutz (2013), Figure 25-1.

Instances of Classes

Class Objects (C)
- Instance factories
- Composed of objects
- Provide behavior that is inherited by all instances
- *Blueprint of an Object*

Instance Objects (I1, I2, ...)
- Concrete items in a program's domain
- Record data that vary per specific object
- *Objects holding data*



Lutz (2013), Figure 25-1.

➠ **Instances inherits attributes from its class**

Writing and Using Python Classes

Python Statements

Table 10-1. Python statements

Statement	Role	Example
Assignment	Creating references	`a, b = 'good', 'bad'`
Calls and other expressions	Running functions	`log.write("spam, ham")`
print calls	Printing objects	`print('The Killer', joke)`
if/elif/else	Selecting actions	`if "python" in text:` `  print(text)`
for/else	Iteration	`for x in mylist:` `  print(x)`
while/else	General loops	`while X > Y:` `  print('hello')`
pass	Empty placeholder	`while True:` `  pass`
break	Loop exit	`while True:` `  if exittest(): break`
continue	Loop continue	`while True:` `  if skiptest(): continue`
def	Functions and methods	`def f(a, b, c=1, *d):` `  print(a+b+c+d[0])`
return	Functions results	`def f(a, b, c=1, *d):` `  return a+b+c+d[0]`
yield	Generator functions	`def gen(n):` `  for i in n: yield i*2`

global	Namespaces	`x = 'old'`
		`def function():`
		`  global x, y; x = 'new'`

nonlocal	Namespaces (3.X)	`def outer():`
		`  x = 'old'`
		`  def function():`
		`  nonlocal x; x = 'new'`

import	Module access	`import sys`
from	Attribute access	`from sys import stdin`

class	Building objects	`class Subclass(Superclass):`
		`  staticData = []`
		`  def method(self): pass`

try/except/ finally	Catching exceptions	`try:`
		`  action()`
		`except:`
		`  print('action error')`

raise	Triggering exceptions	`raise EndSearch(location)`
assert	Debugging checks	`assert X > Y, 'X too small'`
with/as	Context managers (3.X, 2.6+)	`with open('data') as myfile:` `  process(myfile)`

del	Deleting references	`del data[k]`
		`del data[i:j]`
		`del obj.attr`
		`del variable`

Lutz (2013), Ch. 10, pp330-331

class Statement

```
class className(superclass, ...):
    attribute = value
    def methodName(self, ...):
        self.attribute = value
```

That’s a lot at once,
let’s go through this step by step!

class Statement

```
class className(...):
    ...
```

- ***class*** statement creates and names a *class* object

class Statement

```
class className(...):
    attribute = value
```

- ***class*** statement creates and names a *class* object
- assignments (of names) inside *class* create *attributes*
- *attributes* are inherited object state and behavior

Tutorial

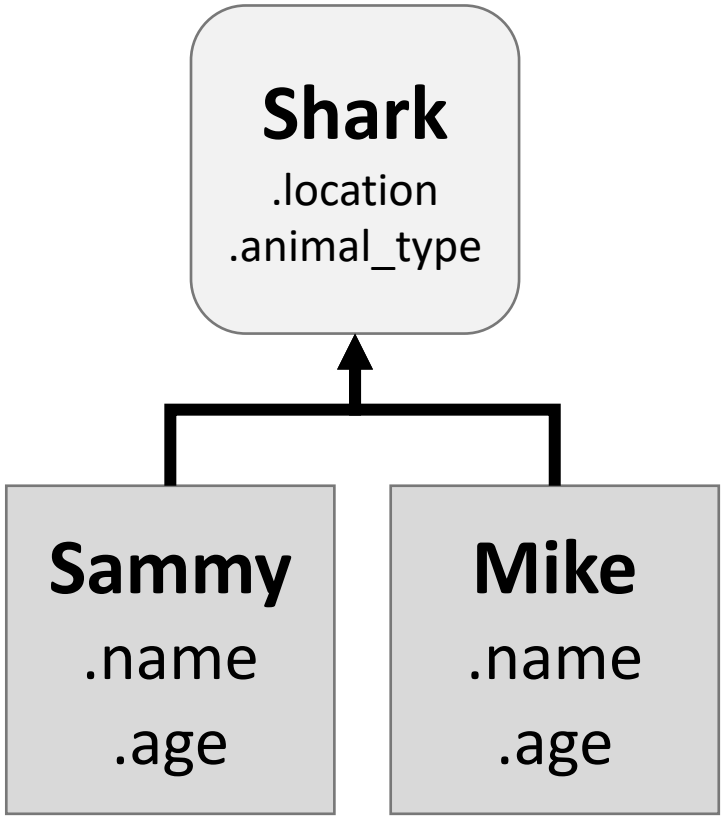
Let’s write a class object

- Write class object of name Shark
- Add class attributes

Shark
.location
.animal_type

How to use a class ?

- “Classes are blueprints for an object”
- Execution only defines the blueprint
- To create an object, need to create **instances of a class**



Lutz (2013), Figure 25-1.

Creating Instances of Classes

```
class className(...):          # class object
    attribute = value          # class variable


instanceVar = className()  # instance object
```

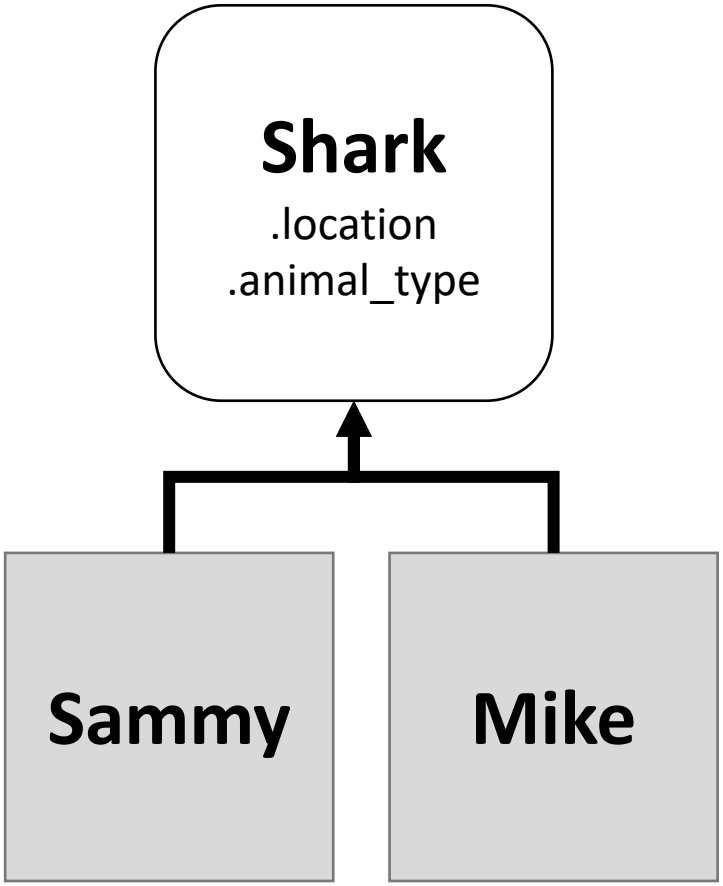
- Each **class** statement creates a **class object** and assigns it a name

- Calling a class object like a function makes a new *instance* object (three objects at this point: two instances, one class)
- Each **instance object** inherits class attributes and gets its own namespace

Tutorial

Create an Instance

- Create two instances of class Shark



Calling Classes Attributes & Methods

```
object.attribute
object.method()
```

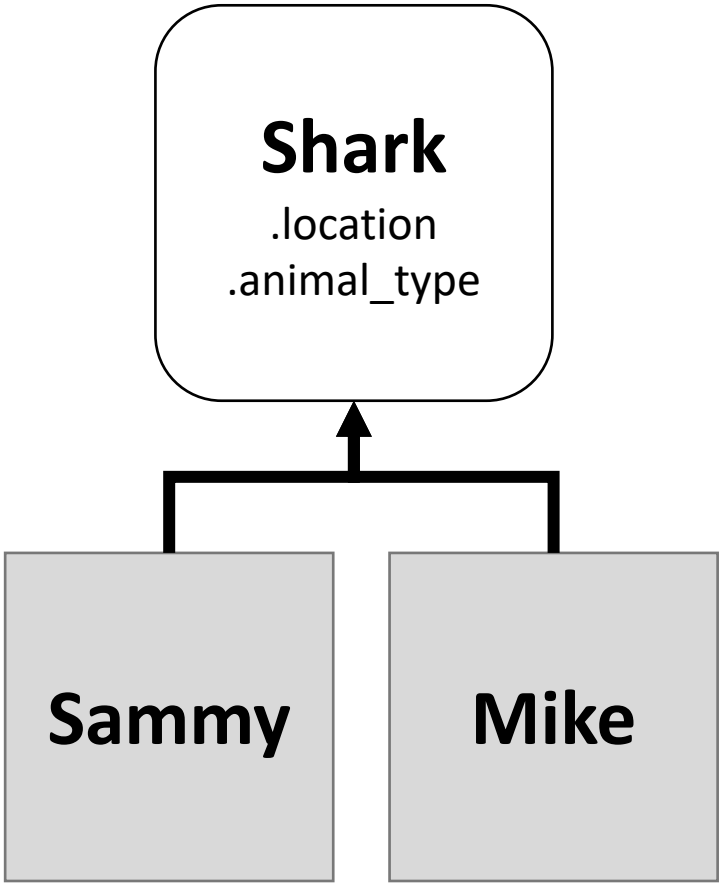
Same dot-notation syntax to call …

- built-in methods of object types: ‘abc’.upper()
- module attributes & functions: math.pi, math.sqrt()
- class objects or class instances: shark.swim()

Tutorial

Call Attributes

- What is the location of Sammy? (Call attributes inherited by the instance)



Writing Class Methods

Enriching the life of your class

class Statement

```
class className(...):

    def methodName(self, ...):
        self.attribute = value
```

- ***class*** statement creates and names a *class* object
- assignments (of names) inside *class* create *attributes*
- *attributes* are inherited object state and behavior
- ***def*** defines class *methods* (function inside a class)
- class *methods* have a special first argument *self*, to receive the implied subject instance

methods' *self* explained

```
class className(...):

    def methodName(self, value, ...):
        self.attribute = value
```

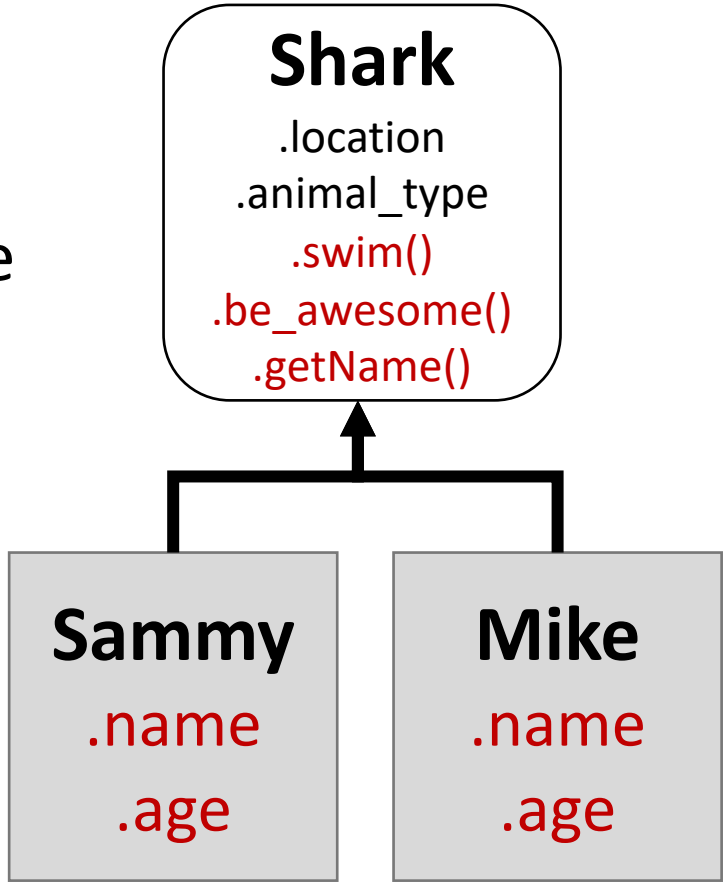
- the first method argument automatically receives an **implied** instance object = subject of the call
- "place holder" for instances = objects created from the class, passed with the dot-operator
- *self* is always first parameter (but not always the only one)
- Without it, *unbound errors* occur.

Tutorial

Add New Methods

- Add simple method
- Call method for an instance
- Experiment with removing *self*

- Add method creating new variables
- Call the method from an instance

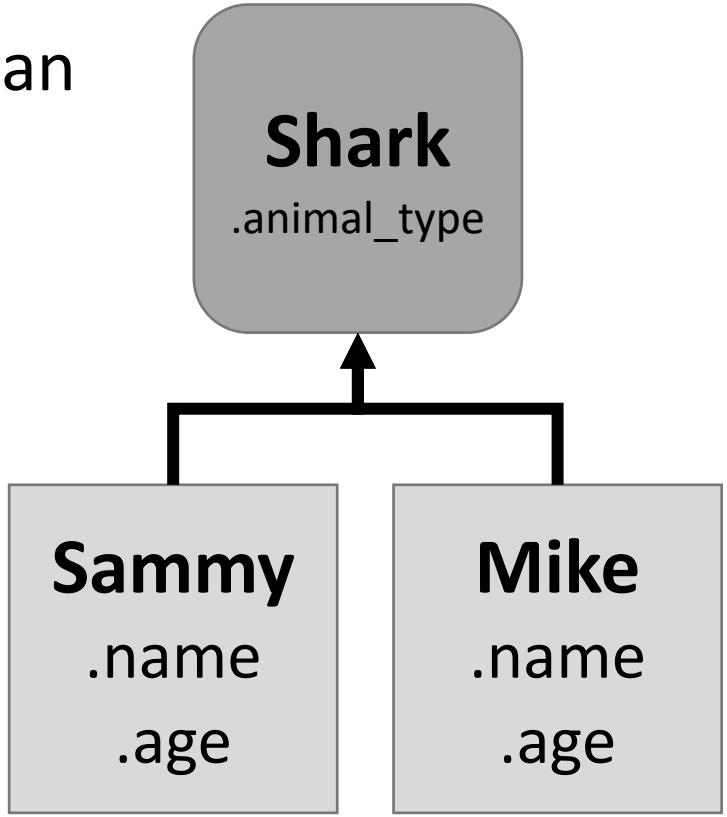


Class & Instance Variables

- Two types of variables that we can define with classes:
  - class variables: class level
  - instance variables: instance level

- **Class variables**
  - consistent across all instances

- **Instance variables**
  - variables that change significantly across instances

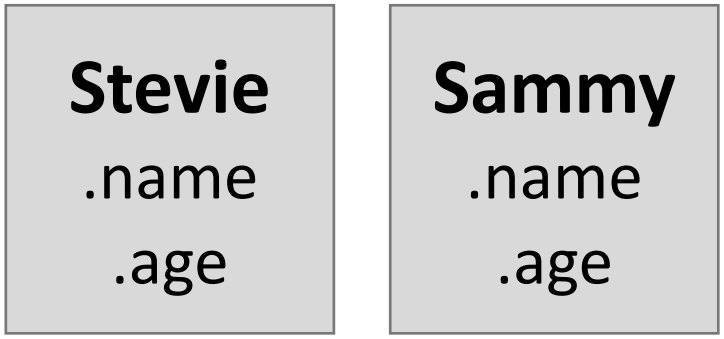


Class Variables

- shared by all instances
- owned by the class

- defined within class construction
- typically placed below the class header (and before the constructor method)

Instance Variables



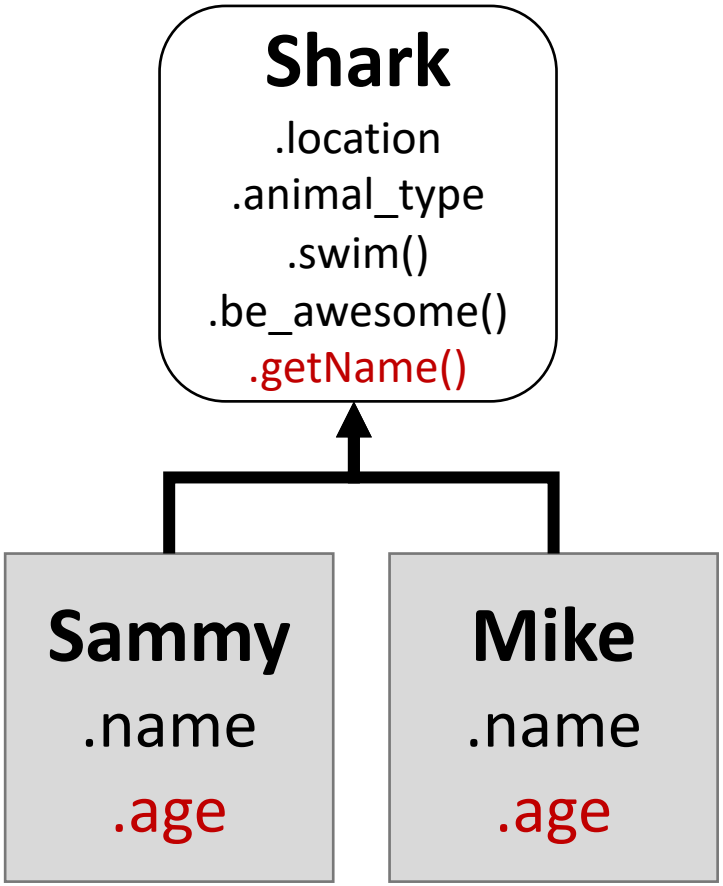
- different for each instance/object
- owned by instances

- defined within **methods**
- created only, when the method is called and does not exist (in the class) before assigned
- Typically created by the **constructor method**

Tutorial

Try it Yourself!

- Add a new variable **age**: it should be added to and printed out in method getName()



Constructor Method : *__init__*

```
class className(...):
    attribute = value

    def __init__(self, attr=default , ...):
        statements
```

- Function of a Class with “special effect”
- Also known as ***__init__ method***
- Constructor does not need to be called
- When a class instance is created, the constructor method is automatically executed (if it was coded or inherited)

Constructor Method : *__init__*

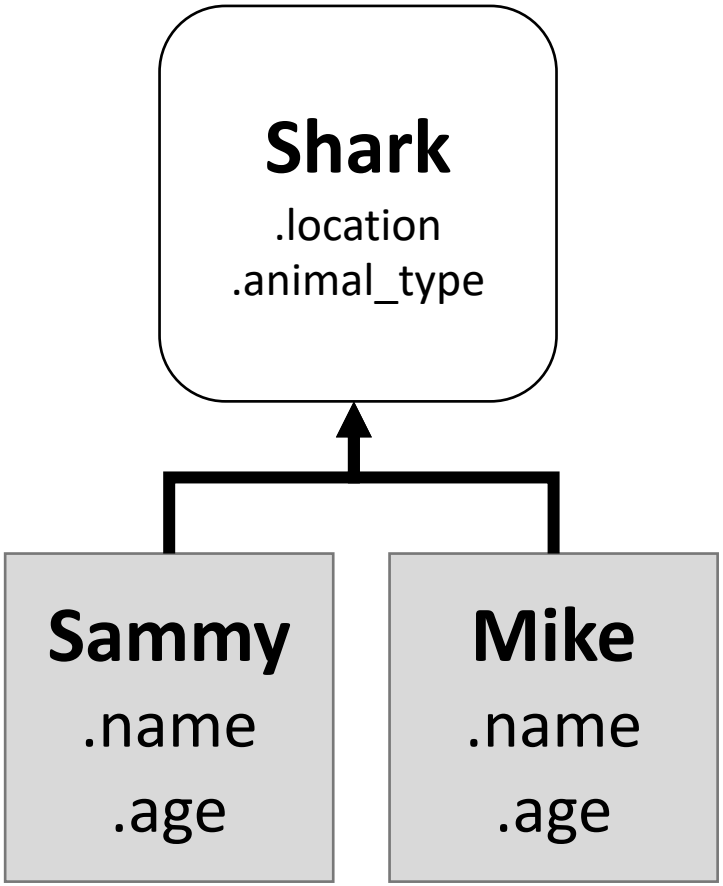
```
class className(...):
    attribute = value

    def __init__(self, attr=default , ...):
        statements
```

- **Initializes data for instances**
- If not present, instances begin life as empty namespaces.
- We can also pass parameter directly to the constructor method

Tutorial

Initiate the Instance

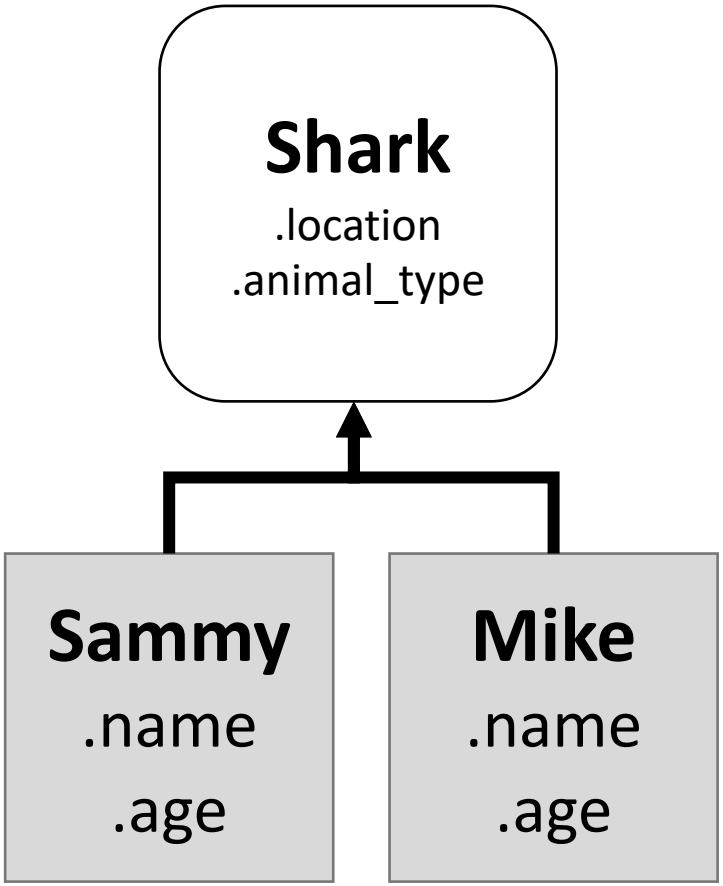
- Add constructor method
- Use it to initialize variables name and age
- Create instance of new Shark class and pass instance variables
- Add default values



Tutorial

Initiate the Instance

- Add another parameter to the constructor method: **age**
- Create a second instance with different content
- Let's swim with friends!



Practice Notes

- L10_reading_OOPipynb
- Functions, Modules and Classes will be part of E05, available next week

- Optional: L10_readingAdvanced_OOPinheritance.ipynb