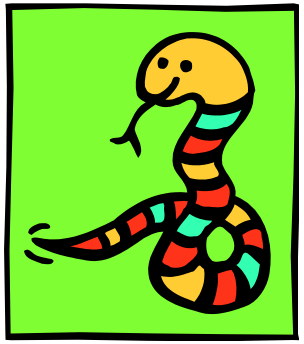
Python Classes

LING 131A, Fall 2019

Marc Verhagen, Brandeis University



```
#!/usr/local/bin/python3

class Animal(object):

    def __init__(self, name, species):
        self.name = name
        self.species = species

    def pretty_print(self):
        print("<Animal name=%s species=%s>" % (self.name, self.species))

class Dog(Animal):

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

    def pretty_print(self):
        print("<Dog name=%s>" % (self.name,))


if __name__ == '__main__':

    a1 = Animal('fluffy', 'dog')
    a2 = Dog('spooky’)
    print(a1)
    print(a2)
    a1.pretty_print()
    a2.pretty_print()
```

Why classes?

- Code organization and code reuse
- Associate functionality with objects of certain types in a natural way
- Object-oriented programming is popular (bad reason for using them, good reason for learning about them)
- But note that there is criticism from several angles
  - Some claiming that functional programming is cleaner
  - Subclassing is not the only way to achieve polymorphism

It's all objects…

- Everything in Python is really an object.
  - We have seen hints of this already…

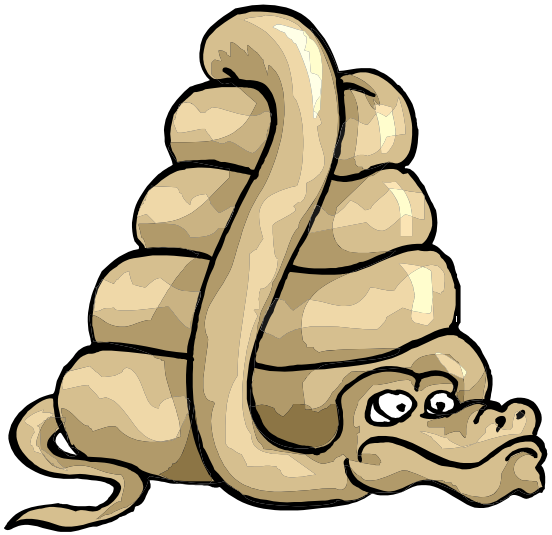
```
"hello".upper()
lst.append('a')
dct.keys()
```

  - You can also design your own object in addition to these built-in data-types.
- In fact, programming in Python is typically done in an object oriented fashion.

Class and instance

- Python classes are objects
- Python makes the distinction between a class/type object and an instance object
- Class/type objects can be used to instantiate instance objects

Defining Classes



Defining a Class

- A *class* is a special data type which defines how to build a certain kind of object.
  - The class also stores some data items that are shared by all the instances of this class. (class variables)
- An *instance* is an object that is created following the definition given inside of the class.
- Python does not use separate class interface definitions as in some languages. You just define the class and then use it.

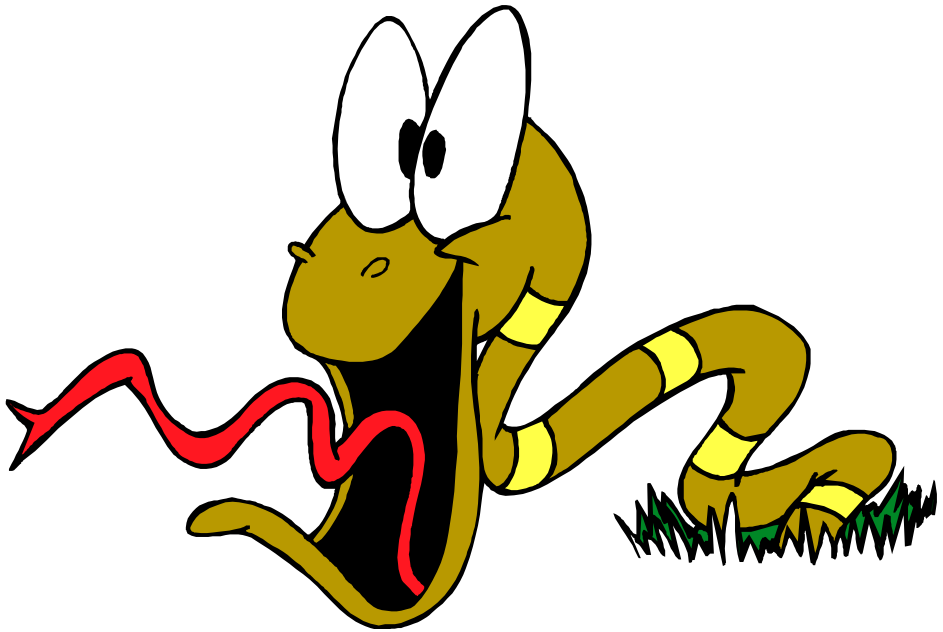
Class (type) and instance

Classes/types	instances
string	“abc”, ‘the’ , ‘world’ , …
list	[1, 2, 3], [‘a’ , ‘b’ , ‘c’]
tuple	(1, 2, 3), (‘a’ , ‘b’ , ‘c’)
dictionary	{ ‘a’ :1,’ b’ :2}
customized class	customized object

Methods in Classes

- You can define a method in a class by including function definitions within the scope of the class block.
  - There is a special first argument `self` in all of the method definitions.
  - There is usually a special method called `__init__` in most classes.

Creating and Deleting Instances



Example class

```
class Student:

    def __init__(self, n, a)
        self.full_name = n
        self.age = a

    def get_age(self):
        return self.age
```

```
class Student:

    def __init__(self, n, a):
        self.full_name = n
        self.age = a

    def get_age(self):
        self.hair = "black"
        return self.age

    def get_hair_color(self):
        return self.hair
```

Constructor: __init__

- `__init__` acts like a constructor for your class.
  - When you create a new instance of a class, this method is invoked. Usually does some initialization work.
  - The arguments you list when instantiating an instance of the class are passed along to the `__init__` method.

    ```
    bob = Student("Bob", 21)
    ```

    So, the __init__ method is passed `"Bob"` and 21.
  - Cannot call it on the class

    ```
    >>> s = Student.__init__("Bob", 21)
    TypeError: __init__() missing 1 required
    positional argument: 'a'
    ```

Constructor: `__init__`

- Your `__init__` method can take any number of arguments.
  - Just like other functions or methods, the arguments can be defined with default values, making them optional to the caller.

- However, the first argument `self` in the definition of `__init__` is special…

Self

- The first argument of every method is a reference to the current instance of the class.
  - By convention, we name this argument `self`.

- In `__init__`, self refers to the object currently being created; in other class methods, it refers to the instance whose method was called.
  - Similar to the keyword **this** in Java or C++.
  - But Python uses **self** more often than Java uses **this**.

Self

- Although you must specify `self` explicitly when defining the method, you don't include it when calling the method.
- Python passes it for you automatically.

Defining a method:

(this code is inside a class definition.)

```
def set_age(self, num):
    self.age = num
```

Calling a method:

```
>>> x.set_age(23)
```

Two versions

```
class Student(object):

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

    def get_name(self):
        return self.name
```

```
>>> john = Student('john')
```

```
>>> john.get_name()
'john'
>>> Student.get_name(john)
'john'
```

These two are the same, the first is syntactic sugar for the second

Random note: No Need to Free

- When you are done with an object, you don't have to delete or free it explicitly.
  - Python has automatic garbage collection.
  - Python will automatically detect when all of the references to a piece of memory have gone out of scope. Automatically frees that memory.
  - Generally works well, few memory leaks.
  - There's also no destructor method for classes (although there is a __del__ method).

Access to Attributes and Methods

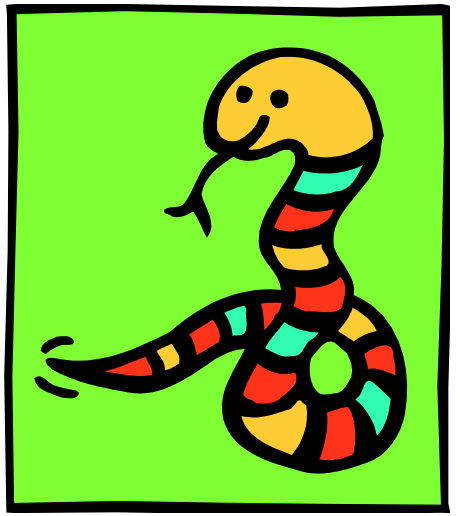
New definition of Student

```
class Student:

    def __init__(self, n, a):
        self.full_name = n
        self.age = a

    def get_age(self):
        self.hair = "black"
        return self.age

    def get_hair_color(self):
        return self.hair
```

Traditional Syntax for Access

```
>>> bob = student ('Bob Smith', 23)
>>> bob.full_name  # Access an attribute.
'Bob Smith'
>>> bob.age        # Access an attribute.
23
>>> bob.hair       # Access an attribute.
??
>>> bob.get_age()  # Access a method.
23
>>> bob.hair()     # Access a method.
??
```

Attributes



Two Kinds of Attributes

- The non-method data stored by objects are called attributes or variables. There are two kinds:
  - **Instance attribute:**
    Variable owned by a particular instance of a class.
    Each instance can have its own different value for it.
    These are the most common kind of attribute.
  - **Class attributes:**
    Owned by the class as a whole.
    All instances of the class share the same value for it.
    Called static variables in some languages.
    Good for class-wide constants or for building counter of how many instances of the class have been made.

Instance Attributes

- You create and initialize a data attribute inside of the `__init__()` method.
  - Remember assignment is how we create variables in Python; so, assigning to a name creates the attribute.
  - Inside the class, you refer to data attributes using `self` – for example, `self.full_name`

```
class Teacher:

    def __init__(self,n):
        self.full_name = n

    def print_name(self):
        print self.full_name
```

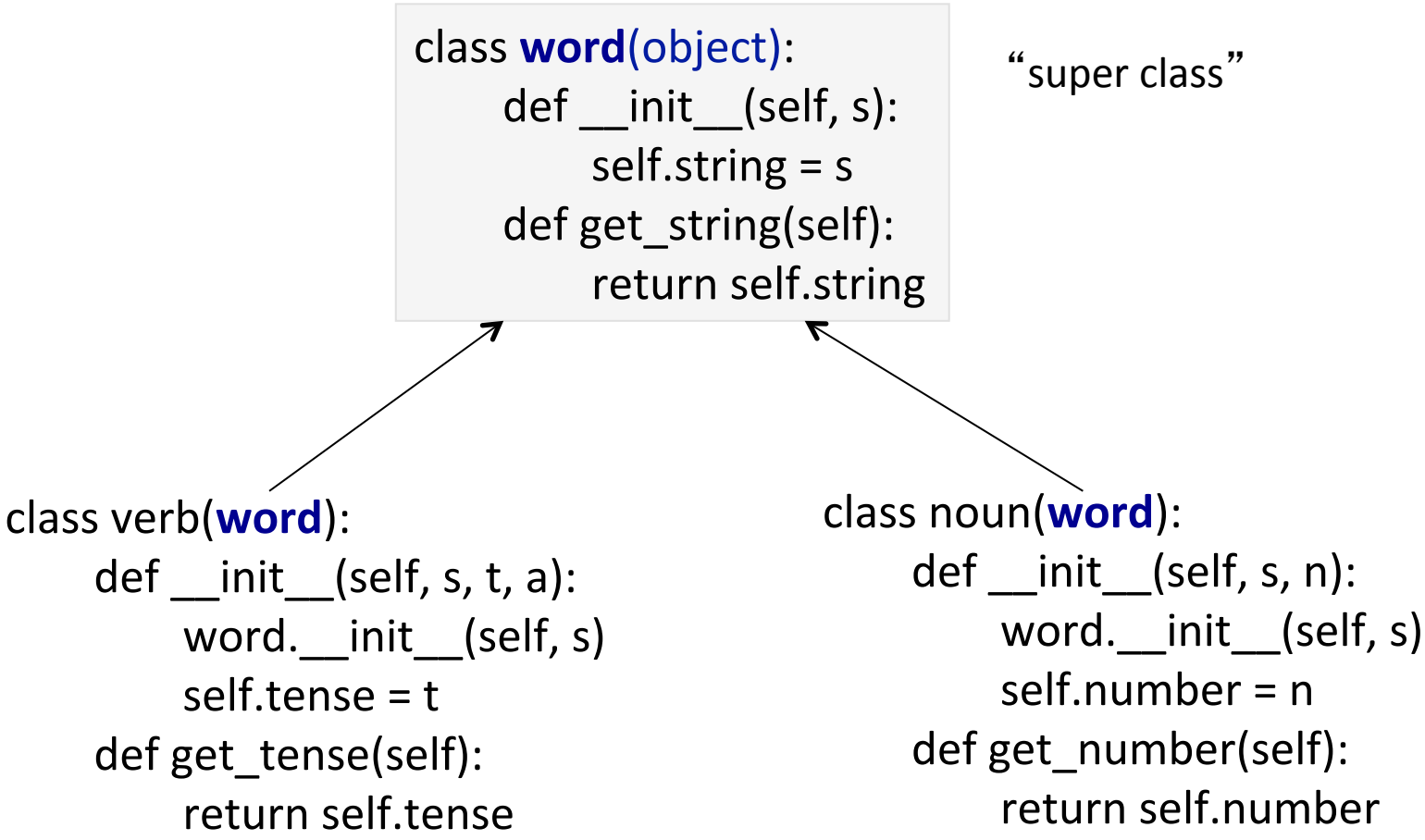
Class Attributes

- All instances of a class share one copy of a class attribute, so when any of the instances change it, then the value is changed for all of them.
- We define class attributes outside of any method.
- Since there is one of these attributes *per class* and not one *per instance*, we use a different notation:
  - We access them using **`self.__class__.name`** notation.

```
class sample:
    x = 23 # class variable
    def increment(self):
        self.__class__.x += 1
```

```
>>> a = sample()
>>> a.increment()
>>> a.__class__.x
24
```

Class inheritance



Class inheritance

- Superclasses are listed in parentheses in a class header
- Classes inherit attributes from their superclasses
- Instances inherit attributes from all accessible classes
- Logic changes are made by subclassing, not by changing superclass

Special methods

- Methods named with double underscore are special hooks
- Such methods are called automatically when instances appear in built-in operations
- Classes may override most built-in operations

Accessing unknown members

- What if you don't know the name of the attribute or method of a class that you want to access until run time?

- Is there a way to take a string containing the name of an attribute or method of a class and get a reference to it (so you can use it)?

getattr(object_instance, string)

```
>>> bob = Student('Bob Smith', 23)

>>> getattr(bob, 'full_name')
'Bob Smith'

>>> getattr(bob, 'get_age')
 <method get_age of class studentClass at 010B3C2>

>>> getattr(bob, 'get_age')()   # We can call this.
23

>>> getattr(bob, 'get_birthday')
AttributeError: Student object has no attribute 'get_birthday'
```

hasattr(object_instance,string)

```
>>> bob = Student('Bob Smith', 23)

>>> hasattr(bob, 'full_name')
True

>>> hasattr(bob, 'get_age')
True

>>> hasattr(bob, 'get_birthday')
False
```

Other topics

- Related to classes and to be introduced at some point…
- Static methods and class methods
- More on magic methods
  - Can be used to let your class emulate strings, numbers, lists, dictionaries and other built-in types
  - https://www.python-course.eu/python3_magic_methods.php

Class methods

- Regular instance methods are associated with an instance of a class

```
>>> fluffy = Dog(fluffy)
>>> fluffy.get_name()
'fluffy'
```

- Class methods are associated with the class itself

```
>>> Dog.get_count()
1
```

```
class Dog(object):

    count = 0

    def __init__(self, name):
        self.name = name
        self.__class__.count += 1

    @classmethod
    def get_count(cls):
        return cls.count


if __name__ == '__main__':
    d1 = Dog('fluffy')
    d2 = Dog('fido')
    print(Dog.get_count())
```

```
#!/usr/local/bin/python3

class Animal(object):

    def __init__(self, name, species):
        self.name = name
        self.species = species

    def pretty_print(self):
        print("<Animal name=%s species=%s>" % (self.name, self.species))

class Dog(Animal):

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

    def pretty_print(self):
        print("<Doc name=%s>" % (self.name,))


if __name__ == '__main__':

    a1 = Animal('fluffy', 'dog')
    a2 = Animal('spooky', 'dog')
    print(a1)
    print(a2)
    a1.pretty_print()
    a2.pretty_print()
```

```
class Animal(object):

    def __init__(self, name):
        self.name = name
        self.can_fly = False
        self.can_sing = False

    def __str__(self):
        return "<%s %s can_fly=%s can_sing=%s>" \
                % (self.__class__.__name__, self.name,
                   self.can_fly, self.can_sing)


class Bird(Animal):

    def __init__(self, name):
        super().__init__(name)
        self.can_fly = True


class Canary(Bird):

    def __init__(self, name):
        super().__init__(name)
        self.can_sing = True
```

Python Classes

```
class Human(object):

    count = 0

    @classmethod
    def population(cls):
        return cls.count

    @staticmethod
    def eats():
        return 'paella'

    def __init__(self, name, beverage):
        self.__class__.count += 1
        self.name = name
        self.beverage = beverage

    def drinks(self):
        return self.beverage
```

Python Classes

- Instance methods
  - Instance is implicitly passed as first argument

```
sue.drinks()
Student.drinks(sue)
```

- Class methods
  - Class is implicitly passed as first argument

```
Student.population()
```

Python Classes

- Static methods
  - No implicit first argument
  - Behave like plain functions except that you can call them from an instance of the class

```
sue.eats()
Student.eats()
```

  - Could just be functions defined outside of the class but may fit logically in the class