Classes

© Global Knowledge Training LLC

Objectives

- Create object-oriented code
- Define classes and inheritance hierarchies
- Create member variables and properties
- Understand object lifecycles
- Override classes' magic methods
- Leverage duck-typing for polymorphic behavior

© Global Knowledge Training LLC

Python 3 classes

- Python 3 is fully object-oriented
- There is a common base class: **`object`**
- Everything is a class
  - strings
  - lists
  - functions
  - exceptions

Python 2.2 introduced “new-style” classes, prior to that were “old-style” classes. Python 3 is a cleaned up implementation of new-style classes.

© Global Knowledge Training LLC

Instantiating existing classes

```
from queue import PriorityQueue

q = PriorityQueue()
q.put("hello queue")
print(q.get())
```

Instantiate classes using:
var = ClassName(args)

Note: no ‘new’ keyword.

Namespace may be required depending on import statement.

```
import queue

q = queue.PriorityQueue()
q.put("hello queue")
print(q.get())
```

© Global Knowledge Training LLC

Defining classes (simple version)

Defined using the **class** keyword

__init__ is the constructor method

self must be passed to every method

Member variables are 'declared' dynamically on self.

Methods are just regular functions within the class

```
class Cat:
    def __init__(self, name, friskiness=50):
        self.name = name
        self.friskiness = friskiness

    def wake_up(self):
        print(self.name +
            "stretches and says 'meeeooow...'")

    def play(self):
        if self.friskiness > 20:
            print(self.name + " begins racing around.")
        else:
            print(self.name + " rolls over.")
```

© Global Knowledge Training LLC

Internals of member variable storage

- Dynamically adding values to **`self`**
  - Calls **`self.__setattr__`**
  - Default implementation is to add to **`self.__dict__`**

self.__dict__	
name	'Fluffy'
friskiness	42

```
class Cat:
    def __init__(self, name, friskiness=50):
        self.name = name
        self.friskiness = friskiness

p = Cat('Fluffy', 42)
```

© Global Knowledge Training LLC

Destructors and cleanup

```
class Cat:
    def __init__(self, name, friskiness=50):
        self.name = name
        self.friskiness = friskiness

    def __del__(self):
        print("deleted, good bye " + self.name)
```

__del__ is the destructor
(not guaranteed to be called if alive during exit)

© Global Knowledge Training LLC

Operator overloading and more

- Classes have many **magic methods**
  - Here are the most important ones
  - See this article for details and many more methods

Method	Purpose
__new__	Object instantiation (rarely used)
__init__	Object constructor (a = new A())
__del__	Called when your object is garbage collected
__exit__	Called when exiting a `with` scope
__eq__	Equality operator (a == b)
__lt__ / __gt__	Less than, greater than operators (a < b)
__str__	String representation for humans (readable)
__repr__	String representation for machines (parsable)
__iter__	Converts your class to be iterable (for s in a: ...)
__len__	The 'length' or count (len(a))
__contains__	Membership check ('pierre' in names)

© Global Knowledge Training LLC

Inheritance [base classes]

Animal is our base (super) class.

Cat derives from Animal

```
class Animal:       # base class
    def __init__(self):
        print("creating animal")

class Cat(Animal): # cat is an animal
    def __init__(self, name, friskiness=50):
        super().__init__()
        self.name = name
        self.friskiness = friskiness
        print("creating cat" + name)
```

```
c = Cat()
# prints
# creating animal
# creating cat
```

Access to the super class methods is via the `super()` method.

Warning: if you don't call **super().__init__()** it will not be called for you!

In Python 2.7, call super like this:
super(Cat, self).__init__()

© Global Knowledge Training LLC

Overriding base methods

```
class Animal:                    # base class
    def wake_up(self):
        print("Animal stretches and wakes up")
```

```
class Cat(Animal):
    def wake_up(self):
        print(self.name +
           "stretches and says 'meeeooow...'")
```

```
class Dog(Animal):
    def wake_up(self):
        super().wake_up()     # invoke base wake_up()
        print(self.name +
           "stretches and says 'whoof...'")
```

Invocation of base method must be explicit

```
c = Cat("Fuffy")
d = Dog("Rover")
c.wake_up()
# Fluffy stretches and says 'meeeooow...'
d.wake_up()
# Animal stretches and wakes up
# Rover stretches and says 'whoof... '
```

© Global Knowledge Training LLC

Polymorphism [duck-typing]

- Python uses duck-typing rather than static typing for compatibility
  - If it walks like a duck, talks like a duck, it is a duck

```
class Computer: # <-- not an animal
    def wake_up(self):
        print("the computer is resuming")

# class Cat(Animal), Animal has wake_up()
cat = Cat("Fluffy")
computer = Computer()

# duck typing
use_animal(cat) # cat says meow
use_animal(computer) # computer resuming

def use_animal(ani):
    ani.wake_up()
```

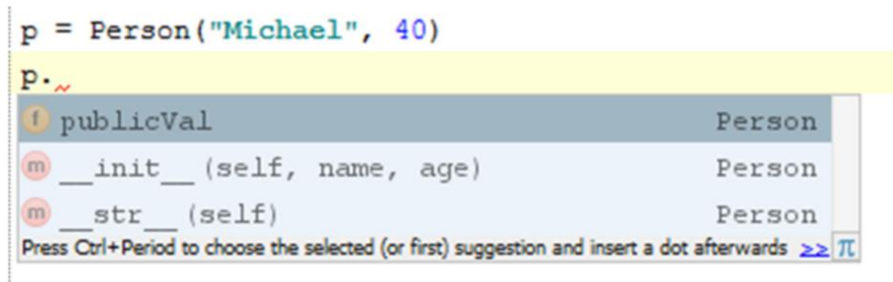
© Global Knowledge Training LLC

Data-hiding and encapsulation [private variables]

- Using **__member** convention limits easy access
  - Access is still possible if you are sneaky (**_Person__name**)

```
class Person:
    def __init__(self, name, age):
        self.__name = name
        self.__age = age
        self.publicVal = "this is public"
```

Only **publicVal** appears in intellisense




```
p = new Person()
print(p.publicVal) # prints this is public
print(p.__name)    # Error!
# AttributeError: 'Person' object has no attribute '__name'
```

This works for both variables and methods.

© Global Knowledge Training LLC

Data-hiding and encapsulation [public properties]

- Encapsulation is possible with **@property** decorator

Create a read-only property called 'name'

Add a setter with validation

```
class Person:
    @property # __name defined in __init__
    def name(self):
        return self.__name

    @name.setter
    def name(self, val):
        if len(val) > 0:
            val = val[0].upper() + val[1:]
        self.__name = val
```

```
p = new Person("Michael", 40)
print(p) # prints Michael is 40
p.name = "ted"
print(p) # Ted is 40
```

© Global Knowledge Training LLC

Static methods

- Classes can have static methods using **@staticmethod**

```
class Person:
    @staticmethod
    def from_JSON(jsonText): # No self argument
        p = Person()
        # set values
        return p

jeff = Person.from_JSON("{name: 'Jeff'}")
type(jeff) # prints <class Person>
```

© Global Knowledge Training LLC

Classes as dynamic objects

- Custom classes can be used dynamically
  - difficult for class to know its values

```
p = Person("Michael", 40)
p.hobbies = ["Biking", "Skiing"] # this defines hobbies
p.hobbies.append("Motocross")

print(p.hobbies)
# prints ['Biking', 'Skiing', 'Motocross']

print(p)
# prints Michael is 40
```

© Global Knowledge Training LLC

Classes as anonymous objects

- Anonymous objects are convenient local classes
  - dictionaries almost fulfil this role
  - we can do better

```
d = dict(name="Michael", age=40)
print(d)          # prints {'age': 40, 'name': 'Michael'}
print(d["name"]) # prints Michael
print(d.name)     # ERROR!
```

© Global Knowledge Training LLC

Classes as anonymous objects

- Combining what we have seen adds anonymous objects
  - classes
  - inheritance
  - magic methods

```
class Anon(dict):
    __getattr__, __setattr__ = dict.get, dict.__setitem__

a = Anon(name="Michael", age=40)
print(a)                # prints {'age': 40, 'name': 'Michael'}
print(a["name"])        # prints Michael
print(a.name)           # prints Michael
a.hobbies = ["Biking"] # calls __setattr__ -> dict.__setitem__
a.hobbies.append("Motocross") # calls __getattr__ -> dict.get
print(a) # prints
{'name': 'Michael', 'age': 40, 'hobbies': ['Biking', 'Motocross']}
```

© Global Knowledge Training LLC

Summary

- Classes are defined with the class keyword
- Member variables (attributes) are added dynamically in the __init__ method
- Properties act like data with validation
- Classes have many magic methods which control their behavior
- Duck-typing allows flexible uses of objects

© Global Knowledge Training LLC