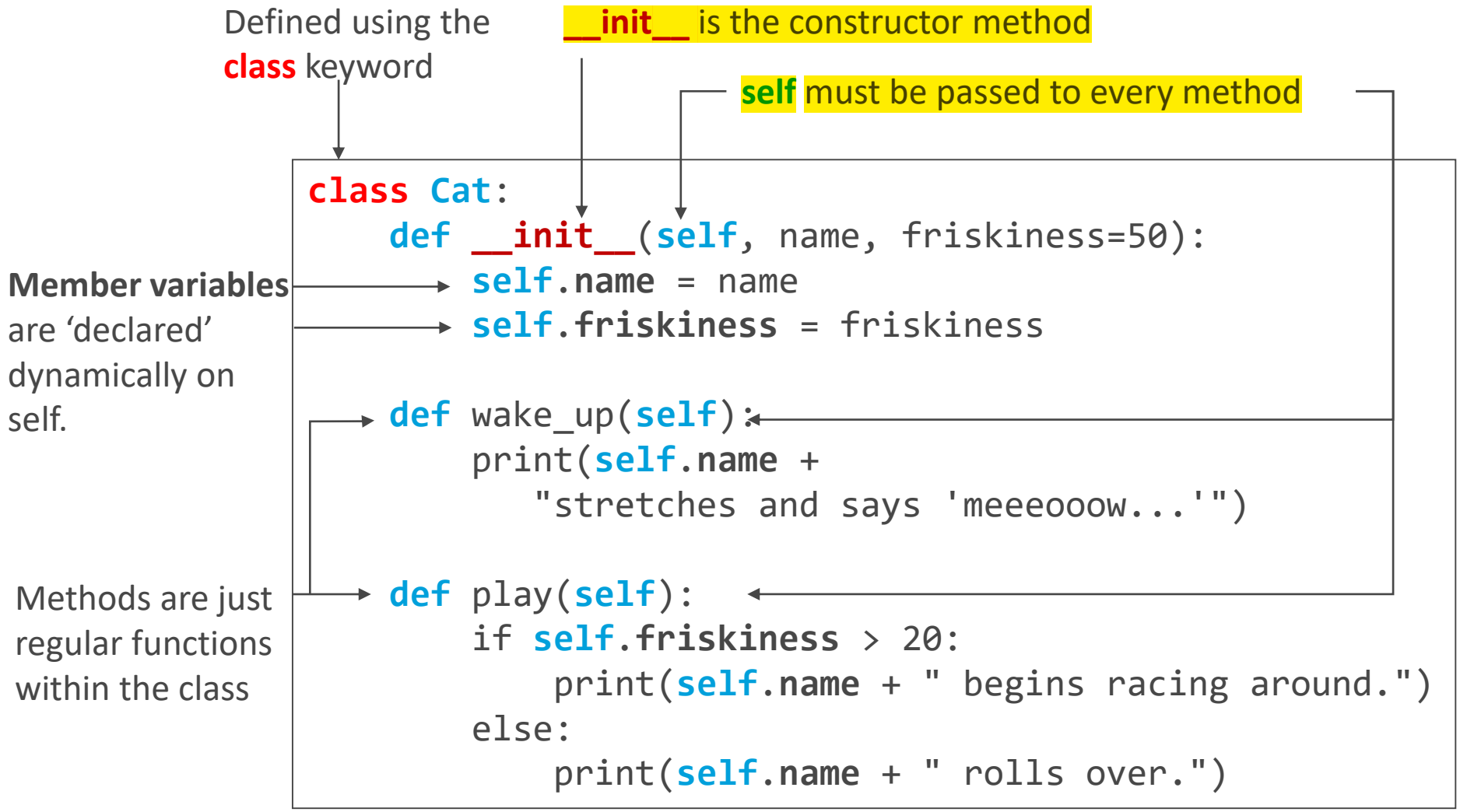
Classes

- 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

Python 3 classes

- Python 3 is fully object-oriented
- There is a common base class: **`object`**

Defining classes (simple version)



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

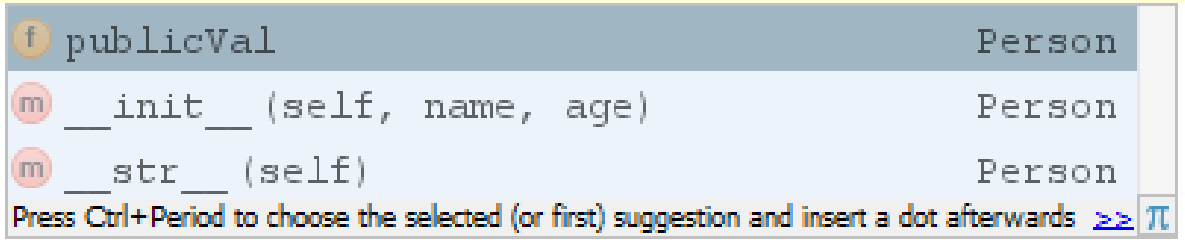
Data-hiding and encapsulation [private variables]

- 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 = Person("Michael", 40)
p.
```




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

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

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)
```

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!

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

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... '
```

Polymorphism

- 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")

cat = Cat()
computer = Computer()

def use_animal(ani):
    ani.wake_up()


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

```
for ani in (cat,computer):
    ani.wake_up()
```

Static methods

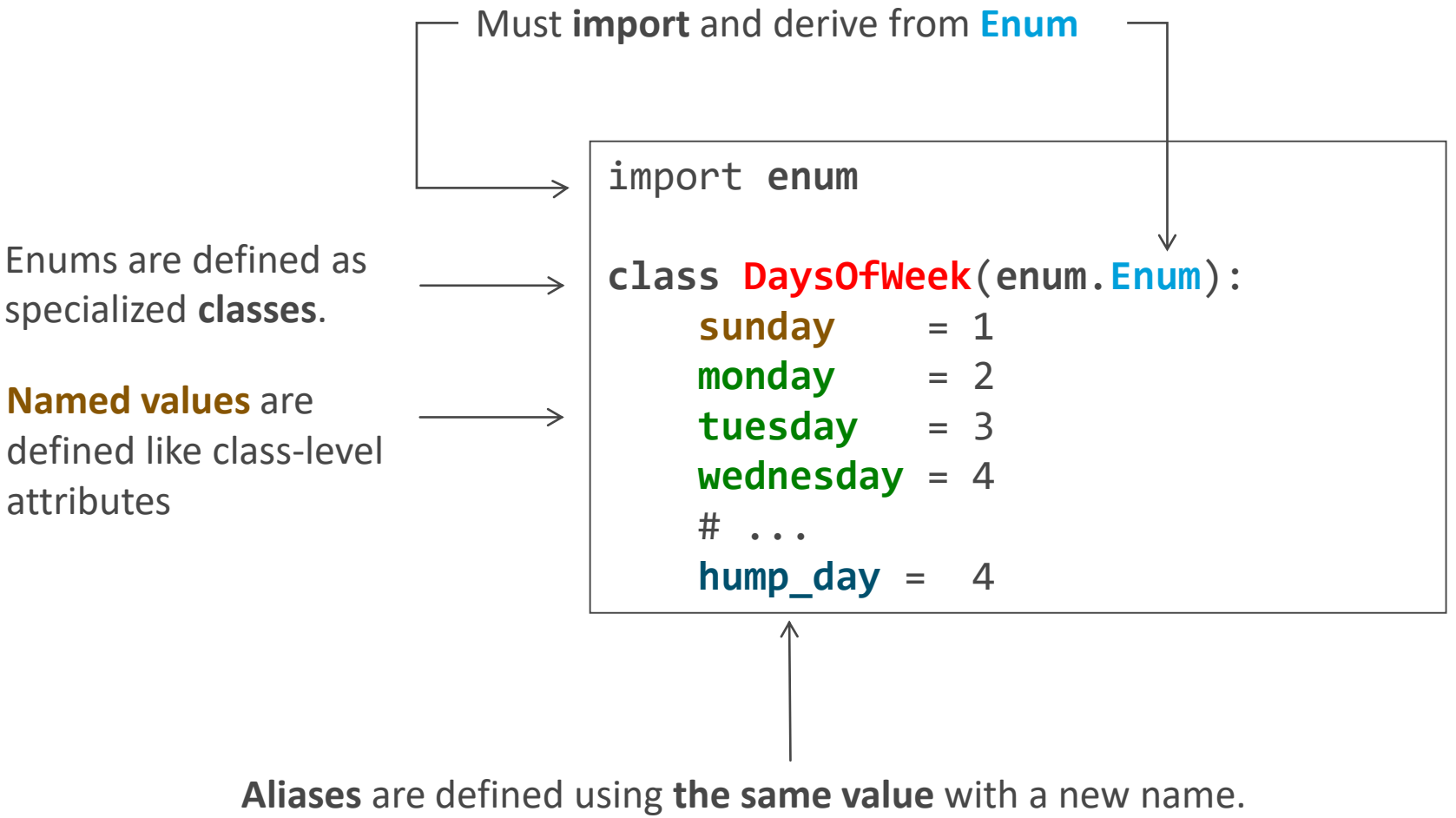
- Classes can have static methods using **@staticmethod**

```
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>
```

Enumerations: Bounded sets of named values

- Python 3.4 added formal **enums** to language



Enumerations: Usage

```
day = DaysOfWeek.wednesday
print('Today is {0}'.format(day))
# Today is DaysOfWeek.wednesday

print("Full name: {0}, readable name: {1}".format(day, day.name))
# Full name: DaysOfWeek.wednesday, readable name: wednesday

if day == DaysOfWeek.saturday or day == DaysOfWeek.sunday:
    print("It's the weekend!")
else:
    print("Work day...")
# Work day...
```

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