Introduction to Python Programming

15 – Object Orientation: Loose Ends

Josef van Genabith (Stefan Thater)
Dept. of Language Science & Technology
Universität des Saarlandes

WS 2021/22

Using implicit superclass

- Instead of referencing the superclass explicitly, you can use `super()`. This always calls the class closest in the inheritance tree.
- Allows instance calls, i.e. does not require explicitly handing over `self`. Advantage: Cleaner, ensures inheritance hierarchy is followed.

```
class CheckingAccount(Account):
  def __init__(self, num, person, credit_range):
    super().__init__(num, person)
    self.credit_range = credit_range
  ...
```

Using implicit superclass

- Instead of referencing the superclass explicitly, you can use `super()`. This always calls the class closest in the inheritance tree.
- Allows instance calls, i.e. does not require explicitly handing over `self`. Advantage: Cleaner, ensures inheritance hierarchy is followed.

```
class CheckingAccount(Account):
  def __init__(self, num, person, credit_range):
    Account.__init__(self, num, person)
    self.credit_range = credit_range
  ...
```

Compare with

Multiple Inheritance

- In some OOP language (e.g. Java) classes can only extend a single class.
- In Python, a class can inherit from more than one class
⇒ Multiple Inheritance
- Special mechanisms to resolve which class’s method is called
- Recommendation: for now, let your classes have at most one superclass.

Everything is an Object

- Unbeknownst to most of us …
- We have used objects in Python right from the start of this course
- Lists and Dictionaries are objects
- Even strings and numbers are objects!

```
>>> n = 123
>>> n.__mod__(2) # is the same as n%2
1
>>>
```

Everything is an Object

- We can therefore subclass the built-in classes

```
class TalkingDict(dict):
   def __init__(self):
       print("Create a new dictionary...")
       dict.__init__(self)
       print("Done!")
   def __setitem__(self, key, value):
       print("Setting", key, "to", value)
       dict.__setitem__(self, key, value)
       print("Done!")

myDict = TalkingDict()
myDict["x"] = 42
```

hook for dict[key] = value

Objects as Keys in Dicts

```
>>> class Person:
...     def __init__(self, name, age):
...         self.name = name
...         self.age = age
...
>>> s1 = Person("John", 26)
>>> d = dict()
>>> d[s1] = 'some value'
>>> d[s1]
'some value'
>>> s2 = Person("John", 26)
>>> d[s2]
KeyError: <__main__.Person object at 0x10076cf28>
```

Two different objects representing John. Python does not know that these two objects refer to the same person. More technically: the hash values of s1 and s2 are different!

Objects as Keys in Dicts

```
>>> class Person:
...     def __init__(self, name, age):
...         self.name = name
...         self.age = age
...     def __hash__(self):
...         return hash(self.name) * hash(self.age)
...     def __eq__(self, other):
...         return self.name == other.name and self.age == other.age
...
>>> s1 = Person("John", 26)
>>> d = dict()
>>> d[s1] = 'some value'
>>> d[s1]
'some value'
>>> s2 = Person("John", 26)
>>> d[s2]
'some value'
```

Lists as Keys in Dicts

```
>>> class MyList(list):       # subclass list
...     def __hash__(list):
...         return id(list)
...
>>> a = MyList([1,2,3])
>>> b = MyList([1,2,3])
>>>
>>> d = dict()
>>> d[a] = 'a'
>>> d[a]
'a'
>>> d[b]
KeyError: [1, 2, 3]            # elements are different
```

Lists as Keys in Dicts

```
>>> class MyList(list):            # subclass list
...     def __hash__(self):        # plus some other stuff
...         h = 1                  # that computes the hash
...         for item in self:      # from the items
...             h *= hash(item)
...         return h
...
>>> a = MyList([1,2,3])
>>> b = MyList([1,2,3])
>>>
>>> d = dict()
>>> d[a] = 'a'
>>> d[a]
'a'
>>> d[b]                           # now we can use them as
'a`                                # keys in dicts …
```

Lists as Keys in Dicts

```
>>> a = MyList([1,2,3])
>>> b = MyList([1,2,3])
>>>
>>> d = dict()
>>> d[a] = 'a'
>>> d[a]
'a'
>>> d[b]
'a'
>>> a.append(4)
>>> d[a]
KeyError: [1, 2, 3, 4]
>>> d[b]
KeyError: [1, 2, 3]
>>> list(d.items())
[([1, 2, 3, 4], 'a')]
```

Static methods

Does not work! Why?

```
class PersistantDict(dict):
  @staticmethod
  def from_file(filename):
    data_from_file = ...
    return PersistantDict(data_from_file)

class PersistantDefaultDict(PersistantDict):
    def __getitem__(self, key):
        ...

pdd = PersistantDefaultDict.from_file("example")
```

Class methods

Works now

```
class PersistantDict(dict):
  @classmethod
  def from_file(cls, filename):
    data_from_file = ...
    return cls(data_from_file)

...
```

Duck Typing

- *When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck.*
- ⇒ we’re not interested in the object’s type or its inheritance from a particular class
- ⇒ we’re interested in the object’s properties, i.e., methods (attributes)