Object-oriented programing – Basic principles

- A system consists of a set of objects that are send messages to each other.
- The reception of a message triggers an operation in the receiving object.
- An object is an individual entity with a unique identity.
- A class describes a set of objects with common characteristics:
  - Attributes (e.g., name, age of a person)
  - Relationships to other objects (e.g., a person is married to another person)
  - Operations that can be executed (e.g., printInfo)
Object-oriented programming – Basic principles

- The current attribute values (and relationships) at a time determines the object's state
- The current state of all existing objects at a time (and their relationships to other objects) determine the system's state
- Classes can be specialized – e.g., an employee is a person
- Fundamental OO concepts
  - Encapsulation
    - Hides particular details
  - Abstraction (inheritance)
    - An “employee” can be regarded as a “person”
  - Polymorphism
    - Behavior dependent on a particular instance
Each payment subclass is handled differently

- Cash Payment
- Credit Payment
- Check Payment

Additionally, the superclass is justified by common attributes and associations.
superclass justified by common attributes and associations

Figure from slides of C. Larman: http://www.craiglarman.com/wiki/index.php?title=Educator_Resources
Classes and objects

- Class ~ (in broad view) a template for creating objects
- Object ~ an instance of a class

In Python
- class defined as a set of statements

```
class ClassName:
    <statement-1>
    .
    .
    .
    <statement-N>
```

- Note – in Python, a class definition is also an object
  - will be later in more details
class Dog:
    kind = 'canine'

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

    def bark(self):
        print(f'{self.name} says: Woof woof')

print(Dog.kind)       # -> canine
d = Dog('Fido')       # instantiating new objects
e = Dog('Buddy')
print(d.kind)         # -> canine
print(e.kind)         # -> canine
print(d.name)         # -> Fido
print(e.name)         # -> Buddy
d.bark()              # Fido says: Woof woof
e.bark()              # Buddy says: Woof woof
Basics of classes

- Method calls

```python
d = Dog('Fido')
Dog.bark(d)  # equivalent to  d.bark()
```

- Calling methods like functions

```python
dbark = d.bark
dbark()
```

- Class variables – shared among all instances

- Object variables defined in `__init__()`
  - but can be defined in any method
  - or even outside of any method

Examine and run `methods_variables.py`
Basics of classes

- Functions can be “transformed” to methods
  
  ```python
  def f1(self, x, y):
      return x + y
  
  class C:
      f = f1

  def g(self):
      return 'hello world'

  h = g

  # now, all f, g, and h are methods
  ```

- functions and methods are objects too
  - will be later in more detail
Inheritance

Methods can be overridden

- effectively, all the methods are virtual (like in Java)
- calling a method from the parent in the overridden method
  \[\text{BaseClassName}\.\text{methodname}(\text{self, arguments})\]
- or (and better)
  \[\text{super()}\.\text{methodname}(\text{arguments})\]

Builtin functions

- \text{isinstance(obj, clazz)}
- \text{issubclass(clazz, parent_class)}
Multiple inheritance

class DerivedClassName(Base1, Base2, Base3):
    <statement-1>
    .
    .
    .
    <statement-N>

- Searching a method/variable in parents
  - generally depth-first, left-to-right

Examine and run multiple_inheritance_basics.py

Not completely true ... details will follow
Inheritance

- All classes inherit (directly or indirectly) from `object`

- Good practice (especially with multiple inheritance)
  - Always call inherited `__init__()` method
    - all of them
  - `super().__init__()`

Examine and run
multiple_inheritance_bad.py
And
multiple_inheritance_ok.py
Linearization

- Searching a method/variable in parents
  - uses C3-linearization (aka Method Resolution Order – MRO)
  - ordering of ancestors such that:
    - ancestor never comes before a child (local precedence order)
    - an ancestor is not visited twice
  - within those rule it builds the MRO depth-first, left-to-right

Examine and run linearization.py
The slides are licensed under a Creative Commons Attribution-NonCommercial 4.0 International License.