Introduction to Python
Part 3: Object-Oriented Programming

Alastair Burt, Christian Federmann
Andreas Eisele, Ulrich Schäfer, Torsten Marek

Universität des Saarlandes

October 7th, 2009
Today’s Topics:

1. Why and When (not) to Use OOP
2. Some Terminology
3. Defining Classes
4. Defining Methods
5. How to Inherit from Classes
6. Some Examples
Part I

Motivation of Object-Oriented Programming
Object-Oriented Programming...

- provides a clear modular structure for programs which makes it good for defining abstract datatypes where implementation details are hidden and the unit has a clearly defined interface.
- makes it easy to maintain and modify existing code as new objects can be created with small differences to existing ones.
- provides a good framework for code libraries where supplied software components can be easily adapted and modified by the programmer. This is particularly useful for developing graphical user interfaces.
Object-Oriented Programming in Python is...

- supported in a very elegant, lightweight fashion
- used in many libraries that come with Python
- generic: built-in objects are often powerful enough → no need to define own object classes
- optional: don’t use it if not useful!
- lightweight and flexible: only use the parts you like, mix and match
You will learn how to...

- define classes and create objects
- define methods
- inherit from existing classes
- overwrite methods and add new ones
You will learn how to...

- define classes and create objects
- define methods
- inherit from existing classes
- overwrite methods and add new ones

Course Focus (reminder)

- What can be done, and how
- very little . . .
  - . . . terminology and theory
  - . . . implementation/machine details
  - . . . comparison to other programming languages
Define an Empty Class and Create an Instance

```python
$ python
>>> class Empty(object): "do nothing"
...
>>> e=Empty()
>>> e
<_main__.Empty instance at 0x00BB34B8>
```
Defining and Using Classes

Define an Empty Class and Create an Instance

```python
$ python
>>> class Empty(object): "do nothing"
...
>>> e=Empty()
>>> e
__main__.Empty instance at 0x00BB34B8
```

Why should we want to do this?

- Objects are containers
- Attributes can be added freely
- Objects have their own namespace
- Objects behave like a dict, but keys need to be names
Define an Empty Class and Create an Instance

```python
>>> e
__main__.Empty instance at 0x00BB34B8
>>> e.a=1
>>> e.a
1
>>> f=Empty()
>>> f.a
AttributeError: Empty instance has no attribute 'a'
>>> dir(e)
['__doc__', '__module__', 'a']
```
Define an Empty Class and Create an Instance

```python
>>> e
<_main__.Empty instance at 0x00BB34B8>
>>> e.a=1
>>> e.a
1
>>> f=Empty()
>>> f.a
AttributeError: Empty instance has no attribute 'a'
>>> dir(e)
['__doc__', '__module__', 'a']

Observations

- Minimal class definition needs only one line
- We create instances by calling a class like a function
- We can use `dir()` to find the attributes of an object
- There needs to be more to it...
Defining and Using Classes

$ python
>>> class Simple(object):
...     def foo(self): return 42
>>> s=Simple()
>>> s.foo
<bound method Simple.foo of __main__.Simple instance at 0x00BB34B8>
>>> s.foo()
42
>>>
Defining and Using Classes

Defining a Method

```
$ python
>>> class Simple(object):
...    def foo(self): return 42
>>> s=Simple()
>>> s.foo
<bound method Simple.foo of <__main__.Simple instance at 0x00BB34B8>>
>>> s.foo()
42
>>> 
```

Observations

- Methods are defined like functions within a class definition
- They need a special first argument ’self’ in the definition
- When calling the method, the instance will fill this argument slot
Defining and Using Classes

''__init__'' is called automatically for new instances

```python
>>> class Counter(object):
...     def __init__(self):
...         self.counter=0
...     def increment(self, inc=1):
...         self.counter += inc
...
>>> c=Counter(); c.increment(); c.increment(10)
>>> c.counter
11
>>> 
```
Defining and Using Classes

'__init__' is called automatically for new instances

```python
>>> class Counter(object):
   ...   def __init__(self):
   ...     self.counter=0
   ...   def increment(self, inc=1):
   ...     self.counter += inc
   ...

>>> c=Counter(); c.increment(); c.increment(10)
>>> c.counter
11
>>> 
```

Observations

- `__init__` called after instance creation, **not** a constructor
- Attributes can be accessed and changed from the outside
- Hiding information is difficult and usually not done
What Have We Done?

**Objects?**

In the last two days, we already have made extensive use of objects without spending too much thought.
What Have We Done?

Objects?

In the last two days, we already have made extensive use of objects without spending too much thought.

What do we know?

- instantiate new objects
- use objects in expressions
- call methods on objects
- delete objects
- store objects for later use
Looking at Objects

Object Activities

- instantiation
  
  ```python
  s = set()
  ```
Looking at Objects

Object Activities

- instantiation
  
  \[ s = \text{set}() \]

- calling methods
  
  \[ s.add(1) \]
Looking at Objects

Object Activities

- instantiation
  \[ s = \text{set}() \]

- calling methods
  \[ s.\text{add}(1) \]

- using objects with operators
  \[ 1 \text{ in } s \]
Looking at Objects

Object Activities

- instantiation
  \[ s = \text{set}() \]

- calling methods
  \[ s.\text{add}(1) \]

- using objects with operators
  \[ 1 \text{ in } s \]

- using generic functions
  \[ \text{len}(s) \]
Looking at Objects

Object Activities

- instantiation
  \[ s = \text{set}() \]
- calling methods
  \[ s.\text{add}(1) \]
- using objects with operators
  \[ 1 \text{ in } s \]
- using generic functions
  \[ \text{len}(s) \]
- creating a new reference
  \[ \text{foo} = s \]
Looking at Objects

Object Activities

- instantiation
  
  ```python
  s = set()
  ```

- calling methods
  
  ```python
  s.add(1)
  ```

- using objects with operators
  
  ```python
  1 in s
  ```

- using generic functions
  
  ```python
  len(s)
  ```

- creating a new reference
  
  ```python
  foo = s
  ```

- deleting a reference
  
  ```python
  del s
  ```
Variables ≡ References ≡ Names

In Python, every variable is a reference to an object (i.e. a piece of memory). An object can have any number of references.
Objects and Variables

Variables ≡ References ≡ Names

In Python, every variable is a reference to an object (i.e. a piece of memory). An object can have any number of references.

== – Equivalence

- about objects (i.e. memory)
- true if the objects have the same content
Objects and Variables

Variables ≡ References ≡ Names

In Python, every variable is a reference to an object (i.e. a piece of memory). An object can have any number of references.

== – Equivalence

- about objects (i.e. memory)
- true if the objects have the same content

is – Identity

- about variables (i.e. references)
- true if two variables refer to the same object
### Object Lifecycle

<table>
<thead>
<tr>
<th></th>
<th>Ref</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>s = set()</td>
<td>1</td>
<td>new instance</td>
</tr>
</tbody>
</table>

**Method/function Calls**

- During the method/function call, the reference count increases by 1.
- Another reference may be stored somewhere else.

**Object Lifecycle**

- **s = set()**: new instance
- **ls = len(s)**: method/function call
- **s2 = s**: new reference
- **del s**: deleted reference
- **s3 = s2.copy()**: object copy
- **del s2**: no more references, destruction!
**Object Lifecycle**

<table>
<thead>
<tr>
<th></th>
<th>Ref</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>s = set()</code></td>
<td>1</td>
<td>new instance</td>
</tr>
<tr>
<td><code>ls = len(s)</code></td>
<td>1</td>
<td>method/function call*</td>
</tr>
</tbody>
</table>
## Object Lifecycle

<table>
<thead>
<tr>
<th></th>
<th>Ref</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>s = set()</td>
<td>1</td>
<td>new instance</td>
</tr>
<tr>
<td>ls = len(s)</td>
<td>1</td>
<td>method/function call*</td>
</tr>
<tr>
<td>s2 = s</td>
<td>2</td>
<td>new reference</td>
</tr>
</tbody>
</table>

*Method/function Calls during the method/function call, the reference count increases by 1 another reference may be stored somewhere else*
Object Lifecycle

| s = set()   | 1 | new instance      |
| ls = len(s) | 1 | method/function call* |
| s2 = s      | 2 | new reference     |
| del s       | 1 | deleted reference |
### Object Lifecycle

<table>
<thead>
<tr>
<th>s = set()</th>
<th>Ref</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>ls = len(s)</td>
<td>1</td>
<td>new instance</td>
</tr>
<tr>
<td>s2 = s</td>
<td>2</td>
<td>new reference</td>
</tr>
<tr>
<td>del s</td>
<td>1</td>
<td>deleted reference</td>
</tr>
<tr>
<td>s3 = s2.copy()</td>
<td>1</td>
<td>object copy</td>
</tr>
</tbody>
</table>

*During the method/function call, the reference count increases by 1.

Another reference may be stored somewhere else.
## Object Lifecycle

<table>
<thead>
<tr>
<th></th>
<th>Ref</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>s = set()</td>
<td>1</td>
<td>new instance</td>
</tr>
<tr>
<td>ls = len(s)</td>
<td>1</td>
<td>method/function call*</td>
</tr>
<tr>
<td>s2 = s</td>
<td>2</td>
<td>new reference</td>
</tr>
<tr>
<td>del s</td>
<td>1</td>
<td>deleted reference</td>
</tr>
<tr>
<td>s3 = s2.copy()</td>
<td>1</td>
<td>object copy</td>
</tr>
<tr>
<td>del s2</td>
<td>0</td>
<td>no more references, destruction!</td>
</tr>
</tbody>
</table>
Object Lifecycle

<table>
<thead>
<tr>
<th></th>
<th>Ref</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>s = set()</code></td>
<td>1</td>
<td>new instance</td>
</tr>
<tr>
<td><code>ls = len(s)</code></td>
<td>1</td>
<td>method/function call*</td>
</tr>
<tr>
<td><code>s2 = s</code></td>
<td>2</td>
<td>new reference</td>
</tr>
<tr>
<td><code>del s</code></td>
<td>1</td>
<td>deleted reference</td>
</tr>
<tr>
<td><code>s3 = s2.copy()</code></td>
<td>1</td>
<td>object copy</td>
</tr>
<tr>
<td><code>del s2</code></td>
<td>0</td>
<td>no more references, destruction!</td>
</tr>
</tbody>
</table>

Method/function Calls

- during the method/function call, the reference count increases by 1
- another reference may be stored somewhere else
Name Lookup in Statically Compiled Languages

In Java etc., all types, members and variables need to be known at compile time, or the compilation will fail.
Comparison

Name Lookup in Statically Compiled Languages

In Java etc., all types, members and variables need to be known at compile time, or the compilation will fail.

In Python

If we can create and delete names and references at any time, how does name resolution work in Python?
Dynamic Name Lookup

Name Resolution at Runtime

In Python, all name resolution happens at runtime, i.e. only and exactly when the program is run and the interpreter encounters a name in the bytecode.
Dynamic Name Lookup

Name Resolution at Runtime

In Python, all name resolution happens at runtime, i.e. only and exactly when the program is run and the interpreter encounters a name in the bytecode.

Unknown Variables

The usage of an unknown variable results in a NameError.
Dynamic Name Lookup

Name Resolution at Runtime

In Python, all name resolution happens at runtime, i.e. only and exactly when the program is run and the interpreter encounters a name in the bytecode.

Unknown Variables

The usage of an unknown variable results in a NameError.

Unknown Members

The try to get an unknown member (i.e. attribute) from an object result in an AttributeError.
Part II

Writing Classes
More Syntax, Finally

The Simplest Class, Ever

class Empty(object):
    pass

e = Empty()
The Simplest Class, Ever

class Empty(object):
    pass

e = Empty()
Class Members

Adding New Members

Methods and fields (class variables) for a class are written into the class definition body:

```python
class NP(object):
    phrase = ['green', 'ideas']
    def length(self):
        return len(self.phrase)
```

```
The Instance Reference: self

Explicit Instance Reference

- first argument to every instance method
- named self by convention
  ...but it’s a pretty strong convention!
- passed in automatically
- always needed to access members of the instance
Comparison to C++/Java

C++/Java: this

- pro self
  - shadowing impossible
  - instance member usage always visible
Comparison to C++/Java

C++/Java: this

- pro self
  - shadowing impossible
  - instance member usage always visible
- contra self
  - easily forgot, leads to unnecessary errors
  - more typing
  - unnecessary difference
Adding a Constructor

To add a constructor to the class, create a method with the name `__init__`:

```python
class Token(object):
    def __init__(self, t):
        self.t = t
...
```
The Constructor Method

Adding a Constructor

To add a constructor to the class, create a method with the name `__init__`:

```python
class Token(object):
    def __init__(self, t):
        self.t = t
...```

Is One Constructor Always Enough?

- default values for method arguments
- use of factory methods encouraged
**Encapsulation**

**No Privacy**

- no access control in classes, everything is "public"
- members starting with "_" are considered private
- trivial getters / setters not needed, and also avoided
- support for name mangling, only rarely used
Encapsulation

No Privacy

- no access control in classes, everything is "public"
- members starting with "_" are considered private
- trivial getters / setters not needed, and also avoided
- support for name mangling, only rarely used

Don’t Abuse This!
Encapsulation

No Privacy

- no access control in classes, everything is "public"
- members starting with "_" are considered private
- trivial getters / setters not needed, and also avoided
- support for name mangling, only rarely used

Don’t Abuse This!

- use it to work around a bug etc. in another class
Encapsulation

No Privacy

- no access control in classes, everything is "public"
- members starting with "_" are considered private
- trivial getters / setters not needed, and also avoided
- support for name mangling, only rarely used

Don’t Abuse This!

- use it to work around a bug etc. in another class
- ... but **only** if you can’t fix it in the code
Inheritance in Python

Some Facts

- for Java programmers: no "final" methods
- for C++ programmers: every method is virtual
- no real abstract methods
- a class can inherit from several base classes
- by default, classes should inherit from object
Inheritance in Python

Some Facts

- for Java programmers: no "final" methods
- for C++ programmers: every method is virtual
- no real abstract methods
- a class can inherit from several base classes
- by default, classes should inherit from object

Legacy Classes

It is possible to create classes that do not inherit from object, but this is only for backwards compatibility with older Python versions.
Inheritance Syntax

Base Class List

The list of base classes is put after the class name:

```python
class File(object):
    ...

class TextFile(File):
    ...

class ImageFile(File):
    ....
```
Part III

Python’s Object System
Polymorphism vs. Inheritance

Inheritance

- hierarchy of classes
- subclasses can inherit functionality
- ... or modify functionality in base classes
- fosters code re-use
Polymorphism vs. Inheritance

Inheritance

- hierarchy of classes
- subclasses can inherit functionality
- ... or modify functionality in base classes
- fosters code re-use

Polymorphism

- possibly unrelated objects provide similar behavior
- need to implement *interfaces* or *protocols*
- existing code uses objects exclusively via interfaces
- fosters generic and extensible code
Examples for Polymorphism

Sequences

- all iterable types (i.e. can be used in a for loop)
- Examples: lists, tuples, dicts, files, sets etc
Examples for Polymorphism

**Sequences**
- all iterable types (i.e. can be used in a for loop)
- Examples: lists, tuples, dicts, files, sets etc

**Input**
- read input from different sources with the same interface
- Examples: files, stdin, network streams, devices, pipes
Recap: Polymorphism in Other Languages

Polymorphism in Java

- *interface* definitions
- an interface groups related methods
- a class can implement any number of interfaces
- implementation of methods enforced by the compiler
Recap: Polymorphism in Other Languages

**Polymorphism in Java**
- *interface* definitions
- an interface groups related methods
- a class can implement any number of interfaces
- implementation of methods enforced by the compiler

**Polymorphism in C++**
- classes with abstract virtual functions only
- any number of interfaces
- implementation of methods enforced by the linker
Polymorphism in Python: Protocols

Polymorphism in Python

- Polymorphism in Python is purely based on names
- no additional syntax
- no inheritance needed
- depends on run-time name lookup

Protocols

A protocol lists the names of functions a class must implement in order to support the protocol.

"Duck Typing"

It walks like a duck, it quacks like a duck, it's a duck!
Polymorphism in Python: Protocols

Polymorphism in Python

- Polymorphism in Python is purely based on names
- no additional syntax
- no inheritance needed
- depends on run-time name lookup

Protocols

A protocol lists the names of functions a class must implement in order to support the protocol.
Polymorphism in Python: Protocols

Polymorphism in Python

- Polymorphism in Python is purely based on names
- no additional syntax
- no inheritance needed
- depends on run-time name lookup

Protocols

A protocol lists the names of functions a class must implement in order to support the protocol.

"Duck Typing"

It walks like a duck, it quacks like a duck, it’s a duck!
Examples

File-Like Objects

An object that implements `read`, `write`, `seek` etc. can be used as an argument to a function that expects a file object.
Examples

File-Like Objects
An object that implements read, write, seek etc. can be used as an argument to a function that expects a file object.

Lists and Strings
Lists and strings do not have a common base class, but can be used in the same way in many contexts, because both implement the sequence and the iterator protocol.
Dynamic vs. Static Typing

Pros of Dynamic Typing

- very flexible
- easy to change
- no type casts
- polymorphism is implicit
Dynamic vs. Static Typing

Pros of Dynamic Typing

- very flexible
- easy to change
- no type casts
- polymorphism is implicit

Cons of Dynamic Typing

- all errors happen at runtime
- depends on ”strong testing” discipline (not really a con...)
- high performance impact
Introspection

Definition

- "look into" objects
- read out the lists of object members at runtime
- run-time availability of type information
Introspection

Definition

- "look into" objects
- read out the lists of object members at runtime
- run-time availability of type information

Uses of Introspection

- explore the interface of an object
- access internal state
- access type information
- access help at runtime
Methods for Introspection

Overview

- **dir**: returns the list of members of an object
- **hasattr**: query if an object has a specific attribute
- **getattr**: get the value of an object’s attribute by name
- **isinstance**: checks if an object is an instance of a specific type
- **callable**: returns true if an object is a function
- **str**: should return a useful description of the object
- **repr**: should return a string representation of a class instance
Function Introspection

Available Information About Functions and Methods

- name
- argument count and names
- default values for arguments
- defining module
Function Introspection

Available Information About Functions and Methods

- name
- argument count and names
- default values for arguments
- defining module

Example Usage

- IPython displays useful information about a function
- `func_summary` does roughly the same
"Indirect" Method Calls

Invoking Methods through Introspection

- instance methods are accessible by `getattr`, too
- methods are "just" objects
- method objects can be called!
Passing around Functions

Functions about Functions

- pass functions as arguments
- return other functions
- called "higher-order functions"
A Simple Example: Function Composition

Function Composition

In Mathematics, the function composition operator \( \cdot \) has the following definition:

\[
(f \cdot g)(x) = f(g(x)) \tag{1}
\]
Function Composition in Python

Python Implementation

```python
def compose(f, g):
    def _composed(x):
        return f(g(x))
    return _composed
```

Explanations

takes two functions as arguments
returns a function
f and g are bound to the arguments (closure)
tomorrow will be all about HOFs
Function Composition in Python

Python Implementation

```python
def compose(f, g):
    def _composed(x):
        return f(g(x))
    return _composed
```

Explanations

- takes two functions as arguments
- returns a function
- \( f \) and \( g \) are bound to the arguments (closure)
- tomorrow will be all about HOFs
Nearly Everything Else Is an Object, Too

More First-Class Objects
Nearly Everything Else Is an Object, Too

More First-Class Objects

- types
  - explore classes and built-in types
  - access documentation
  - pass classes to functions
Nearly Everything Else Is an Object, Too

More First-Class Objects

- **types**
  - explore classes and built-in types
  - access documentation
  - pass classes to functions

- **modules**
  - access module-level documentation
  - get a list of module members
Part IV

Advanced OO Features
Is Empty useless?

Filling Empty

We can add arbitrary attributes to the empty class:

```python
p = Empty()
p.surname = "Fry"
p.names = ["Philip", "J."]
p.age = 25
p.job = "delivery boy"
```

Also, existing attributes can be deleted and reassigned.
The Cost of Flexibility

Attribute Lookup

- every attribute access is a dictionary lookup
- small overhead, but can become important if speed is paramount
The Cost of Flexibility

Attribute Lookup

- every attribute access is a dictionary lookup
- small overhead, but can become important if speed is paramount

Wasted memory

- every instance of a class has holds dictionary instance
- uses a lot of space if large numbers of small classes are instantiated
Examples

Some Examples for Operator Overloading in Python

- string and list concatenation
- indexed access with []
- set intersection and union with & and |
- containment for sets, strings, lists and tuples
- "string multiplication"
- comparison and equality checks for strings etc.
- value binding in string formatting

Built-in types are not special with regard to operator overloading!
Overloading Operators

Magic Methods

Operator support in classes is done by implementing magic methods.
Overloading Operators

Magic Methods

Operator support in classes is done by implementing magic methods.

Examples

<table>
<thead>
<tr>
<th>Operation</th>
<th>Operator</th>
<th>Method</th>
<th>Arity</th>
</tr>
</thead>
<tbody>
<tr>
<td>addition</td>
<td>+</td>
<td><strong>add</strong></td>
<td>2</td>
</tr>
<tr>
<td>bitwise xor</td>
<td>^</td>
<td><strong>xor</strong></td>
<td>2</td>
</tr>
<tr>
<td>inversion</td>
<td>~</td>
<td><strong>invert</strong></td>
<td>1</td>
</tr>
</tbody>
</table>
Overloading Operators

Magic Methods

Operator support in classes is done by implementing magic methods.

Examples

<table>
<thead>
<tr>
<th>Operation</th>
<th>Operator</th>
<th>Method</th>
<th>Arity</th>
</tr>
</thead>
<tbody>
<tr>
<td>addition</td>
<td>+</td>
<td><strong>add</strong></td>
<td>2</td>
</tr>
<tr>
<td>bitwise xor</td>
<td>^</td>
<td><strong>xor</strong></td>
<td>2</td>
</tr>
<tr>
<td>inversion</td>
<td>~</td>
<td><strong>invert</strong></td>
<td>1</td>
</tr>
</tbody>
</table>

Right-associative Variants

Binary operators also have a variant if they are on the right of the object they are combined with.
Applications of Operator Overloading

Item Lookup

Look up two words in a two-dimensional cooccurrence matrix:

```
freq = cooc_matrix["python", "snake"]
```
Applications of Operator Overloading

**Item Lookup**

Look up two words in a two-dimensional cooccurrence matrix:

```python
freq = cooc_matrix["python", "snake"]
```

**Matrix Manipulation**

NumPy (a numerics package for Python) defines operators to support matrix manipulation:

```python
>>> ma = array([1, 2])
>>> ma + ma
array([2, 4])
```
Doubtful Examples

Corner-Cases in Operator Overloading

Filesstem Path Library in Boost (C++):

```python
path fullpath = root / filename
```
Doubtful Examples

Corner-Cases in Operator Overloading

Filesystem Path Library in Boost (C++):

```python
path fullpath = root / filename
```

When overloading...

- don’t stray too far from the original meaning of the operator, avoid casual similarities
- try to implement intuitive operators, not clever ones
Method Types

Classes can have member functions that are not instance methods. Examples:

- static methods
- class methods
Method Types

Classes can have member functions that are not instance methods. Examples:

- static methods
- class methods

Changing Method Semantics

The semantics of a method can be changed using a method decorator:

```python
@some_decorator
def dwim(x):
    ...
```
Static Methods

Description

- similar to static methods in Java/C++
- do not receive a `self` argument
- often used for factory methods
- belong to the class conceptually
- equivalent to normal functions
Static Methods: Example

Static Methods as Factory Methods

class X(object):
    @staticmethod
    def from_file(file_obj):
        ...

    @staticmethod
    def new():
        ...

Class Methods

Description

- instead of the instance, class methods get the class object as their first argument
- exist in the class namespace
- inherited by subclasses
Class Methods

Description

- instead of the instance, class methods get the class object as their first argument
- exist in the class namespace
- inherited by subclasses

Use Cases for Class Methods

- modify static class members
- inherit non-standard constructors
We covered only the most basic features.

Additionally, you can:

- inherit from multiple base classes
- inherit from built-in types
- control attribute access
- control class instantiation (for singletons etc.)
- control how classes are assembled
- modify class instances at runtime
- modify existing types at runtime
- create new types at runtime