02
Object-oriented Programming
Chapter 2

Goals, Principles, and Patterns

Objects are instances of classes.

Classes have data members (instance variables), and member functions (methods).

Classes provides an interface to its consumers (public methods and members) while hiding inner working details (private members and methods)

Goals, Principles, and Patterns

Object-oriented Design (OOD) Goals:

Robustness

Software should produce correct expected results. It should be able to handle unexpected inputs.

Adaptability

Software needs to be easily adaptable to changing conditions (Changes in OSes, browsers, hardware, base libraries, etc. should require less "adaptation" work on the software.)

Reusability

Software should be able to be used as a component of different applications or systems.

Goals, Principles, and Patterns

Object-oriented Design (OOD) Principles:

Modularity

"high cohesion, loose coupling (*)"

Closely related functions and classes should be gathered in a module. Interaction among various modules should be **simple** and **clear** through public members and methods.

Modularity

- Establishes a more convenient debugging environment,
- Supports reusability, and
- Enables clear work sharing among development teams.

(*) For a nice read: https://thebojan.ninja/2015/04/08/cohesion-coupling/

Goals, Principles, and Patterns

Object-oriented Design (OOD) Principles:

Abstraction

Define a complex system with its most fundamental parts.

How to define (an example in the data structures domain):

- Name the functionality (e.g., linked list)
- Explain the functionality (only "what"s, not "how"s)
  - The types of data stored
  - Supported operations (public interface) (e.g., add, drop, pop, push, etc.)
  - Parameters of the operations

Abstract Data Types (ADT): Application of abstraction notion to data structures.

Goals, Principles, and Patterns

Object-oriented Design (OOD) Principles:

Abstraction

Python supports abstract data types using abstract base class (ABC) mechanism.

An ABC cannot be directly instantiated, however, concrete classes can inherit from them (More about this later on).

Goals, Principles, and Patterns

Object-oriented Design (OOD) Principles:

Encapsulation

Internal details of a system or module should not be revealed, and should not be relied on by the "consumers" of that module.

Public interface should guarantee the required functionality.

Python supports loose encapsulation support.

Relies on convention that private members and methods should start with underscore (e.g., _priv_value).

Software Development

Software development mainly involves three major steps:

- Design
- Implementation
- Testing and Debugging

Software Development

Design

Define the main functionalities that will be supported by the software. Further divide functionalities into sub-functionalities and classes.

Build class hierarchy.

Decide how classes will interact, what data they will store, what actions they will perform.

Software Development

Design

Responsibilities:

Divide the work into different actors. Make up action verbs for each responsibility (append_item()).

Independence:

Distribute the responsibilities across classes as independently as possible.

Behaviors:

Define behaviors of each class carefully and precisely. The outputs of the methods should be well-defined.

Software Development

Design

Low-fidelity prototyping can be done with Class-Reponsibility-Collaborator (CRC) cards.

http://agilemodeling.com/artifacts/crcModel.htm

Organization of the software can be explained with unified modeling language (UML).

https://www.omg.org/spec/UML/2.5.1/PDF

https://www.comp.nus.edu.sg/~cs2103/AY1516S2/files/UML%20reference%20sheet.pdf

and many more on the Internet.

Pseudo-codes can be used to express algorithms.

Software Development

Coding

Code should be easy to read and understand, and use a style that follows the community's conventions.

The official style guide for Python is PEP-8:

https://www.python.org/dev/peps/pep-0008/

Some take-aways from the guide:

- Indentation matters. Determines the extent of the control blocks.
- Indentations should be made with 4 space characters, not tabs.
- Use identifier names that can be read aloud.

Software Development

Coding

Some take-aways from the guide (continued):

- Class names should be capitalized, camel cased, and singular (e.g., Date, CarWheel OK, date, Dates, carWheel, car_wheel not OK.)
- (Member) functions should be lowercase, have words separated with an underscore, be a verbal phrase. It can be noun for functions returning single value.
- Parameters, instance and local variables should be lowercase noun. Internal members can start with underscore.
- Comment frequently.

Software Development

Documentation

Formal documentation can be directly typed in source code and can be retrieved in a variety of ways.

If a string literal exists as the first statement in the body of a function, class, or module, it will be treated as documentation.

`help(x)` can be used to retrieve the documentation.

`pydoc` tool can be used generate text or web documeent.

For authoring guidelines:

https://www.python.org/dev/peps/pep-0257/

Software Development

Testing

Essential to software development.

Method coverage: All methods of a class is tested at least once

Statement coverage: Each statement is tested at least once

Special cases of the input should be tested (empty lists, etc.)

Top-down testing: Testing proceeds from top to bottom (stubbing).

Bottom-up testing: Testing proceeds from top to bottom (unit testing).

Module testing: `if name == __main__ :`

`unittest` module

Software Development

Debugging

A simple but not scalable technique: Using `print` statements

`pdb` (CLI interface, in standard Python distribution)

IDEs for Pythons (e.g., IDLE) have debuggers with GUI support

Classes

In Python, every value is an instance of a class.

A class provides

- A set of behaviors (member functions), and
- A state of operation represented with a set of attributes (data members).

```
class CreditCard:
    # Body of the class
```

Classes

`self` identifies the instance upon which a method is invoked.

`self` is necessary while referencing class members and methods.

`self` has to be one of the parameters of class methods.

```
class CreditCard:
    # Body of the class
    def get_customer(self):
        """Return name of the customer."""
        return self.customer
```

Classes

`self` identifies the instance upon which a method is invoked.

`self` is necessary while referencing class members and methods.

`self` has to be one of the parameters of class methods.

```
class CreditCard:
    # Body of the class
    def get_customer(self):
        """Return name of the customer."""
        return self.customer

cust = my_card.get_customer() # called without the
first param
```

Classes

Constructor method of a class initializes the state of the newly created instance.

the constructor function

```
class CreditCard:
    def __init__(self, customer, bank, acnt, limit):
        # omitted comments
        self._customer = customer
        self._bank = bank
        self._account = acnt
        self._limit = limit
        self._balance = 0

cc = CreditCard( 'John Doe', '1st Bank' , '5391 0375
9387 5309' , 1000)
```

instance variables

underscore "implies" that this is a private member

Classes

When a `.py` file is interpreted, the statements at level 0 indentation are executed first.

However, preceding the execution, a special identifier `__name__` is initialized:

- `__name__` is a special value that evaluates to the name of current module.
- If a module is run directly by the interpreter, identifier `__name__` is assigned value `'__main__'`
- If a module is being imported, identifier `__name__` is assigned value `'<module_name>'`

Classes

`__name__` Example

```python
# file1.py
a = 'dummy'

#file2.py
import file1

print('value of file1.__name__: %s' %file1.__name__)
print('value of file2.__name__: %s' %__name__)
```

```
In [177]: %run file2.py
value of file1.__name__: file1
value of file2.__name__: __main__
```

imported module

module being run directly

Classes

`__name__` a popular usage: Unit Testing

```
# CreditCard Module
class CreditCard:
    # implementation omitted

# if module is run directly, this block will be executed
if __name__ == '__main__':
    # Perform tests
    cc = CreditCard( 'John Doe', '1st Bank' , '5391 0375 9387 5309' , 1000)
    cc.charge(100)
    #etc...
```

Operator Overloading

Operators such as +, -, etc. are already implemented for built-in classes: `2+3` (for `int`), `'ali' + 'bak'` (for `string`)

Operators are undefined for new classes, by default.

Common Syntax	Special Method Form	
a + b	a.__add__(b);	alternatively b.__radd__(a)
a − b	a.__sub__(b);	alternatively b.__rsub__(a)
a * b	a.__mul__(b);	alternatively b.__rmul__(a)
a / b	a.__truediv__(b);	alternatively b.__rtruediv__(a)
a // b	a.__floordiv__(b);	alternatively b.__rfloordiv__(a)
a % b	a.__mod__(b);	alternatively b.__rmod__(a)
a ** b	a.__pow__(b);	alternatively b.__rpow__(a)
a << b	a.__lshift__(b);	alternatively b.__rlshift__(a)
a >> b	a.__rshift__(b);	alternatively b.__rrshift__(a)
a & b	a.__and__(b);	alternatively b.__rand__(a)
a ^ b	a.__xor__(b);	alternatively b.__rxor__(a)
a | b	a.__or__(b);	alternatively b.__ror__(a)

Operator Overloading

a += b a −= b a *= b ...	a.__iadd__(b) a.__isub__(b) a.__imul__(b) ...
+a	a.__pos__()
−a	a.__neg__()
˜a	a.__invert__()
abs(a)	a.__abs__()
a < b	a.__lt__(b)
a <= b	a.__le__(b)
a > b	a.__gt__(b)
a >= b	a.__ge__(b)
a == b	a.__eq__(b)
a != b	a.__ne__(b)
v in a	a.__contains__(v)
a[k]	a.__getitem__(k)
a[k] = v	a.__setitem__(k,v)
del a[k]	a.__delitem__(k)
a(arg1, arg2, ...)	a.__call__(arg1, arg2, ...)

Operator Overloading

len(a)	a.__len__()
hash(a)	a.__hash__()
iter(a)	a.__iter__()
next(a)	a.__next__()
bool(a)	a.__bool__()
float(a)	a.__float__()
int(a)	a.__int__()
repr(a)	a.__repr__()
reversed(a)	a.__reversed__()
str(a)	a.__str__()

Notes on Operators:

- Some operators may work without any error on user-defined class instances (e.g., `bool(x)` is always true except for None type. If `__len__` is implemented, `bool(x)` will return `true` when `len() > 0`.).

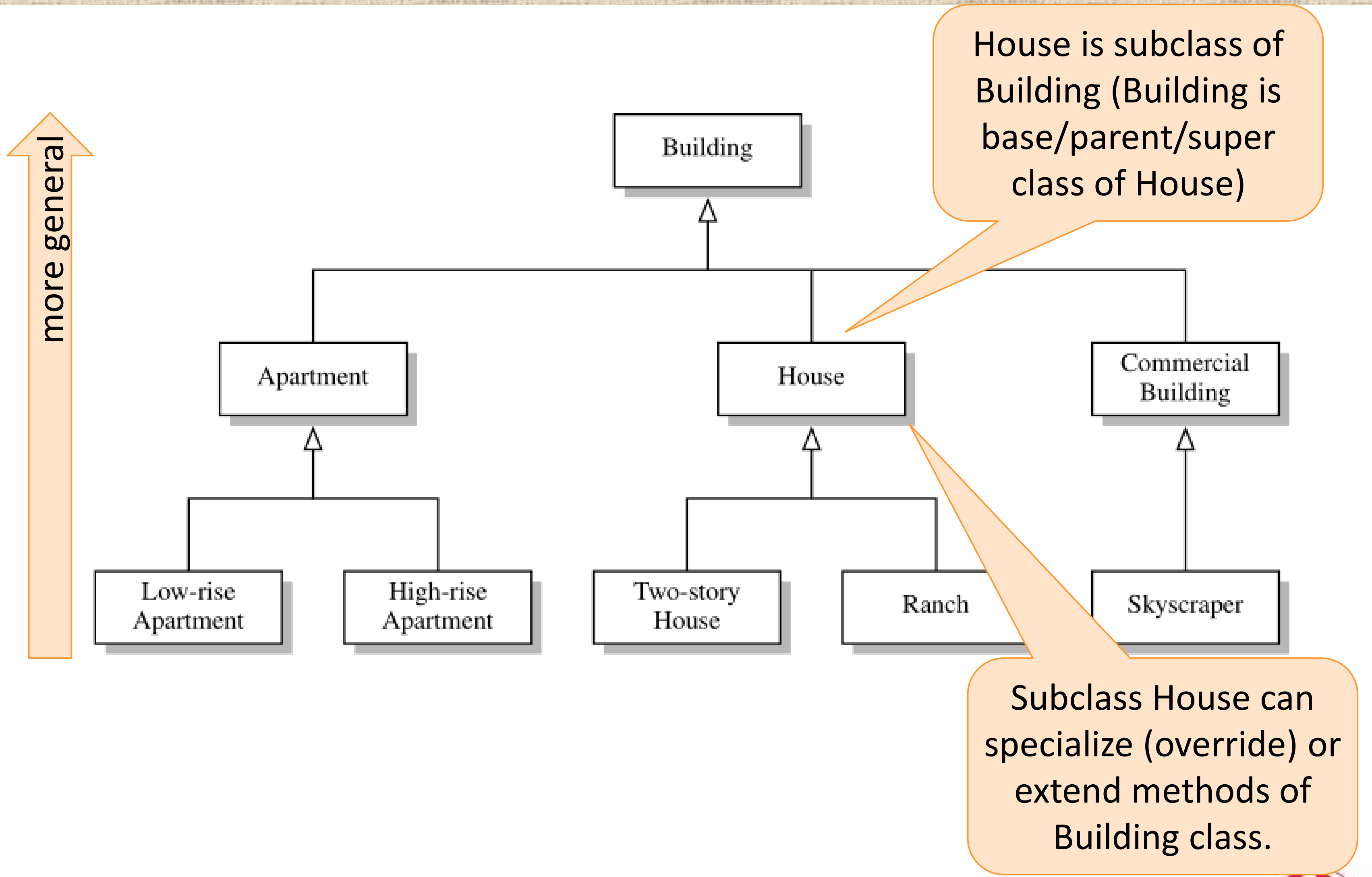
Operator Overloading

Notes on Operators:

- If `__len__` and `__get_item__` are implemented, a default iteration mechanism is provided (i.e., `__iter__` works).
- If `__eq__` is not implemented, `a == b` and `a is b` have the same meaning.
- Implementation of `__eq__` does not support evaluation of `a != b`, to make it work `__ne__` should also implemented separately.

Check Code Fragment 2.4, 2.5, and 2.5 in the reference book.

Inheritance



Inheritance

Inheritance example

Requirements:

A building has an address, and a number of rooms.

A room has a name (an arbitrary string) and a size.

When printed, a building prints the sum of the square footages of all of its rooms.

New rooms can be added to a building.

Example taken from: http://www.cs.toronto.edu/~david/courses/csc148_f14/week2/inheritance.html

Inheritance

```
class Building:
    def __init__(self, address, rooms):
        """ (Building, str, list of Room) -> NoneType """
        self.address = address
        self.rooms = rooms

    def __str__(self):
        """ (Building) -> str """
        sum = 0
        for room in self.rooms:
            sum += room.size
        return str(sum)

    def add_room(self, room):
        """ (Building, Room) -> NoneType """
        self.rooms.append(room)

class Room:
    def __init__(self, name, size):
        """ (Room, str, float) -> NoneType """
        self.name = name
        self.size = size
```

Example taken from: http://www.cs.toronto.edu/~david/courses/csc148_f14/week2/inheritance.html

Inheritance

Inheritance Example (continued)

Additional requirements:

- A house is a type of building with at most 10 rooms.
- Prints the details of all of its rooms (name and square footage, separated by commas).

```python
class House(Building):
    def __init__(self, address, rooms):
        """ (House, str, list of Room) -> NoneType """
        if len(rooms) > 10:
            raise TooManyRoomsError
        else:
            Building.__init__(self, address, rooms)
            # super().__init__(address, rooms) # Another option

    def __str__(self):
        s = 'Welcome to our house\n'
        for room in self.rooms:
            s += '{}, {}\n'.format(room.name, room.size)
        return s
```

```
In [188]: h = House('address', [])

In [189]: str(h)
Out[189]: 'Welcome to our house\n'

In [190]: Building.__str__(h)
Out[190]: 'O'
```

Example taken from: http://www.cs.toronto.edu/~david/courses/csc148_f14/week2/inheritance.html

Abstract Base Classes

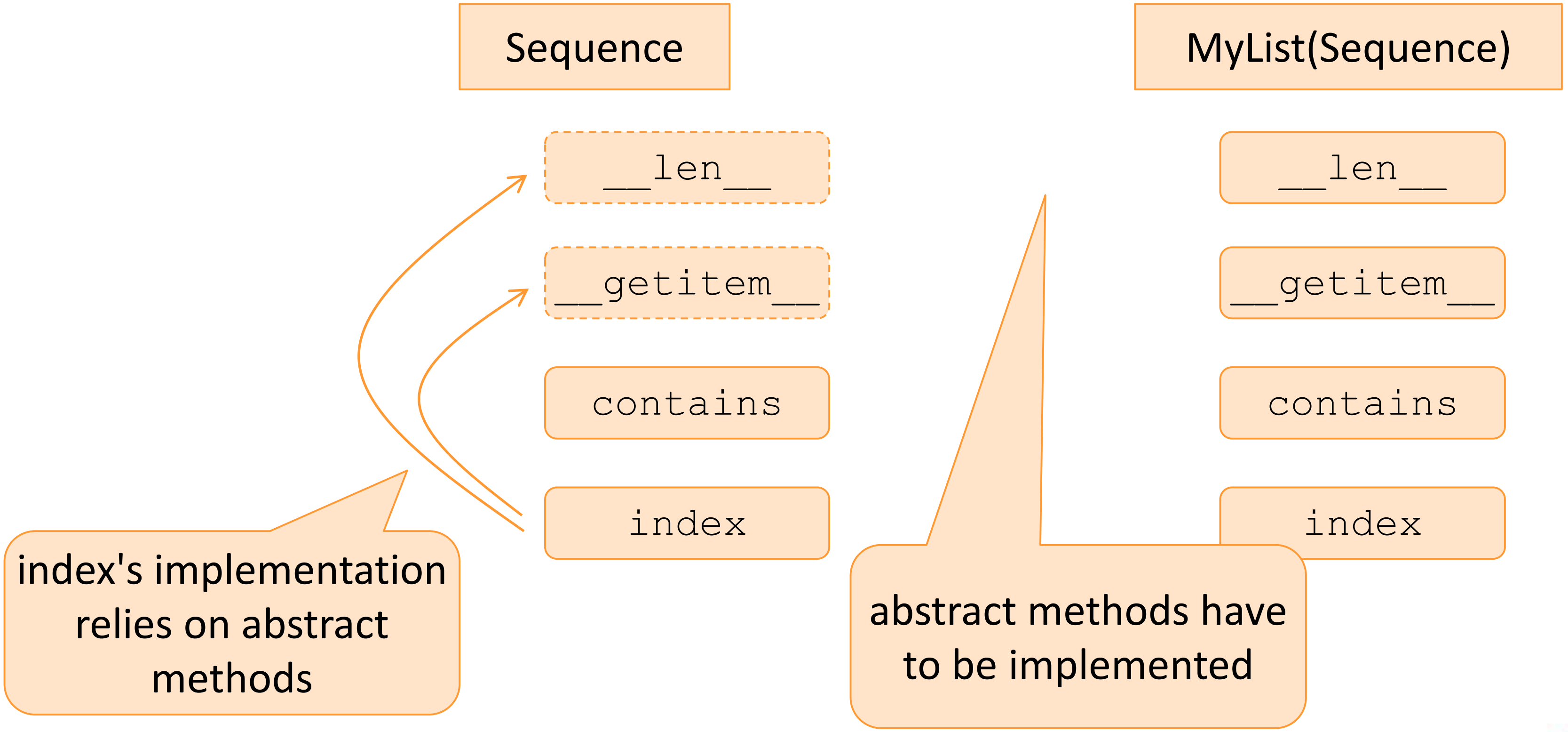
Abstract base classes are structures that can be used as base template classes that can be inherited.

Abstract base classes cannot be instantiated.

Some or all of its methods might not have been implemented prior to been inherited (abstract methods).

Abstract Base Classes

Being one of the OOD pattern, **template method pattern** enables creation of different classes that share common interface.



Abstract Base Classes

a module that "provides infrasturcture for defining abstract base classes" (see PEP 3119 for rationale)

a decorator to identifying abstract methods

a metaclass to create "abstract" base classes, provides a template for the class definition

abstract methods

concrete methods relying on abstract methods

```
from abc import ABCMeta, abstractmethod

class Sequence(metaclass=ABCMeta):

    @abstractmethod
    def len (self):
        """Return the length of

    @abstractmethod
     def getitem (self, j):
     """Return the element at index j of the sequence."""

     def contains (self, val):
     """Return True if val found in the sequence; Fal
        for j in range(len(self)):
            if self[j] == val: # found match
                return True
        return False

     def index(self, val):
     """Return leftmost index at which val is found (or raise ValueError)."""
        for j in range(len(self)):
            if self[j] == val: # leftmost match
                return j
        raise ValueError( value not in sequence ) # never found a match
```

Abstract Base Classes

```
class Range(collections.Sequence):
    # Body of the class
    # Note that we need to have implementations for
    # "len" and "getitem" abstract methods
```

Namespaces and Object-orientation

A class and its instances have their own separate namespaces.

Instance Namespace

Manages attributes specific to an individual object.

Class Namespace

Includes members that are "shared" by all instances.

Such members are not cloned for each instance.

Namespaces and Object-orientation

Class Namespace

A class namespace includes all the identifiers that are declared in class body (e.g., global variables, function definitions, nested classes).

```
class A:
    SOME_GLOBAL_VAR = 1 ✓

    def __init__(self): ✓
        self._a_member_var = 5

    def foo(self):✓
        a_local_var = 'local'
        return a_local_var

    class B:✓# a nested class definition here
        def __init__(self):
            return
```

Namespaces and Object-orientation

Let's explain with examples.

```
$> python.exe
```

> *"Without arguments, return the list of names in the current local scope. With an argument, attempt to return a list of valid attributes for that object."*

```
>>> __name__ # Name of current (global level) module
'__main__'

>>> dir() # names in global level module (__main__ module)
['__annotations__', '__builtins__', '__doc__', '__loader__',
'__name__', '__package__', ....

>>> dir(object) # names in object class which is the base
class for all classes
['__class__', '__delattr__', '__dir__', '__doc__', '__eq__',
'__format__', '__ge__', ...
```

Namespaces and Object-orientation

```
>>> class class_level1:
# consider this as "class class_level1(object): "
...     lvl1_attr = 1
...
...     def __init__(self):
...         self._priv1 = 1999
...
...     def foo():
...         local1 = 111
...         return 1
...
>>> # Class Namespace =
>>> #   All that can be accessed from within class
>>> #   - "object"'s namespace
>>> [x for x in dir(class_level1) if x[0:2] != '__']
['foo', 'lvl1_attr', '__init__']
```

`__init__` is overridden in class's implementation. So, the `__init__` name is different than that of `object` superclass.

Namespaces and Object-orientation

```
>>> class class_level2(class_level1):
...     lvl2_attr = 1
...
...     def __init__(self):
...         self._priv2 = 2999
...
...     def bar():
...         local2 = 222
...         return 2
...
>>> # Class Namespace =
>>> #   All that can be accessed from within class
>>> #   - "object"'s namespace
>>> #   - superclass's namespace
>>> [x for x in dir(class_level2) if x[0:2] != '__']
['bar', 'foo', 'lvl1_attr', 'lvl2_attr', '__init__']
```

`__init__` is overridden in class's implementation. So, the `__init__` name is different than that of `object` superclass.

Namespaces and Object-orientation

Instance Namespace

An instance namespace includes all the identifiers that are directly added to namespace with the `self` identifier.

`self` represents the currently constructed instance.

```
class A:
    SOME_GLOBAL_VAR = 1

    def __init__(self):
        self._a_member_var = 5 ✓

    def foo(self):
        a_local_var = 'local'
        return a_local_var

    class B: # a nested class definition here
        def __init__(self):
            return
```

Namespaces and Object-orientation

```
>>> c1 = class_level1()
>>> [x for x in dir(c1) if x[0:2] != '__']
# class_level1 instance namespace
['_priv1', 'foo', 'lvl1_attr']
```

identifiers from class namespace, not instance's identifiers

```
>>> c2 = class_level2()
>>> [x for x in dir(c2) if x[0:2] != '__']
# class_level2 instance namespace
['_priv2', 'bar', 'foo', 'lvl1_attr', 'lvl2_attr']
```

identifiers from class namespace, not instance's identifiers

identifiers from instance's namespace

Namespaces and Object-orientation

__slots__

Namespaces are managed by making use of a `dict` object.

We may end up having many objects during run time, and in that case that would lead to performance issues.

To alleviate the problem, we can use `__slots__` declaration so that we can streamline instance variables. As a result, a `tuple`, instead of a `dict`, will be used to manage the namespace.

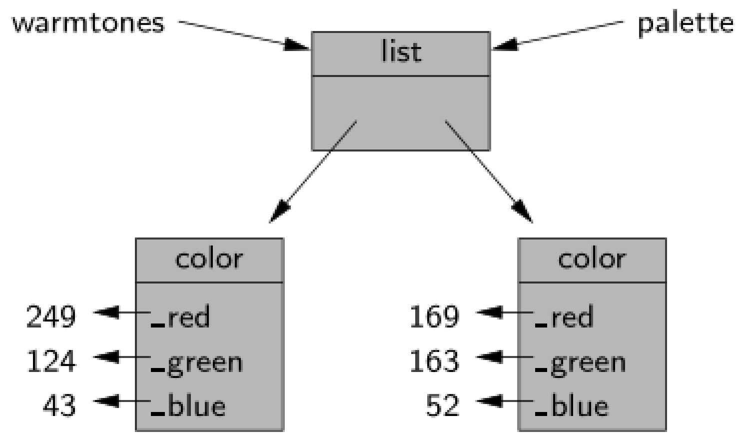
```
class A:
    __slots__ = '_member_var1', '_member_var2', '_member_var3'
```

note that this is a tuple

Shallow and Deep Copying

Creating Alias

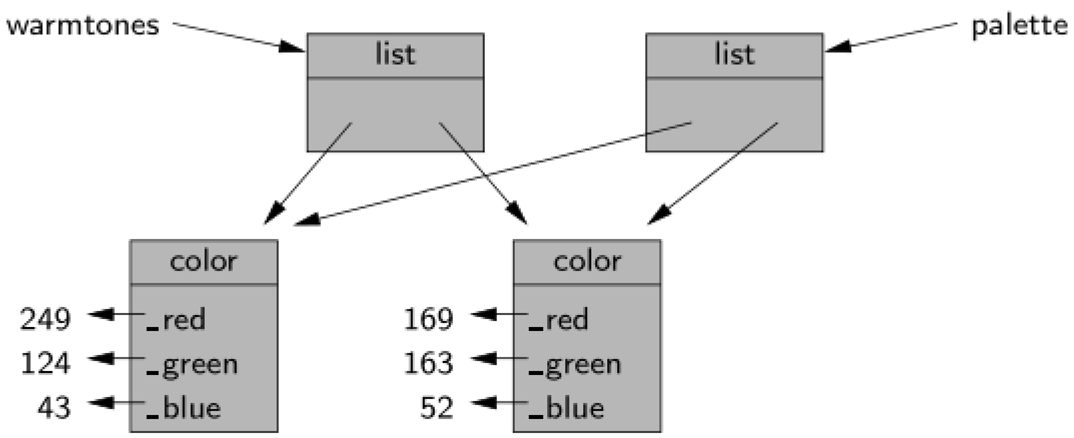
```
palette = warmtones
```



Shallow and Deep Copying

Creating Shallow Copy

```
palette = list(warmtones)
palette = copy.copy(warmtones)
```



Shallow and Deep Copying

Creating Deep Copy

```
palette = copy.deepcopy(warmtones)
```