Object Oriented Programming with Python

Lecture 2

Topics to Cover

- Introduction to OOP - Motivation
- Classes & Objects
- Operator Overloading
- Iterator for Class
- Inheritance
- Abstract Classes
- Name Spaces
- Shallow Vs Deep Copying

Introduction

- *What is Object Oriented Programming (OOP)?*
  - It is an approach for modelling real-world things (e.g., car, house) and relations between things (e.g. student and teacher, company and employee).
  - In other words, OOPs models real-world entities as software objects and governs relationship among them.

- *Why OOP is needed?*
  - OOP aims to provide the following qualities to the software program
    - Robustness against failures
    - Adaptability – Large codes can be easily adapted to accommodate new changes
    - Reusability – Same code base can be re-used in multiple application with little effort

- *How does OOP achieve these Goals?*
  - Encapsulation – By binding data and methods together and limiting its access from outside.
  - Modularity – Different software components are divided into different functional units – Inheritance.
  - Abstraction – Providing a simple and intuitive interface while hiding the implementation details – making it easier for others to understand and use the code - Abstract classes.

Software Development

1. Design
2. Implementation
3. Testing & Debugging
    a. Top-down: stubbing
    b. Bottom-up: unit testing
    c. Debugger - breakpoints, print statements

```
if __name__ == '__main__':
    # perform tests...
```

The Code that is shielded in a conditional construct of the above form will be executed when Python is invoked directly on that module, but not when the module is imported for use in a larger software project.

Class Definitions

- A class serves as the primary means for abstraction in object-oriented programming.

- A class consists of the following two components:
  - Methods or member functions
  - Attributes: Data members, fields or instance variables.

- A class should provide
  - Encapsulation - data members are nonpublic
  - Error Checking
  - Codes for testing a class methods

Classes & Objects

- Class is a blueprint for creating objects.
- It binds data and method together.
- __init__() is the constructor which is called when an object is instantiated.
- Python does not support formal access control.
- It enforces data protection only *by convention*
  - Protected member names starts with single underscore '_' .
  - Private data member names start with double underscores '__'

Variables starting with __ give a error when we try to access them

```python
class Person:
  def __init__(self, name=None, age=None, gender=None):
    self._name = name     # protected or nonpublic
    self._age = age
    self.__gender = gender  # private

  def get_attrib(self):
    print("Hello my name is ", self._name)
    print("My age is", self._age)
    print("My Gender is", self.__gender)

  def set_attrib(self, name, age, gender):
    self._name=name
    self._age = age
    self.__gender = gender

p1 = Person("John", 36, "Male")
p1.get_attrib()
p1.set_attrib("Harry", 25, "Male")
p1.get_attrib()

p2 = Person()
p2.get_attrib()

p2.__gender = "Female"    # still accessible
p2._name = "Sally"        # Still accessible
p2._age = 23              # Still accessible

p2.get_attrib()

print("Gender of P2 is", p2.__gender)
```

```
Hello my name is  John
My age is 36
My Gender is Male
Hello my name is  Harry
My age is 25
My Gender is Male
Hello my name is  None
My age is None
My Gender is None
Hello my name is  Sally
My age is 23
My Gender is None
Gender of P2 is Female
```

Operator Overloading

- Operator overloading: Re-defining the behavior of standard operators and functions for various user-defined objects.

The standard operator '+' provides different functionality for different operands.

```
A = Student1()
B = Student2()
Team = A+B ??
```

```python
# Operator overloading
print(2+3)  # addition of numbers
print([2,3] + [4,5])
a = [[2,3],[4,5]]
b = [[7,8],[10,11]]
c = a + b     # extending an array
print(c)
print("Tom"+"Harry")  # concatenation of strings
```

```
5
[2, 3, 4, 5]
[[2, 3], [4, 5], [7, 8], [10, 11]]
TomHarry
```

Operator Overloading through specially named methods

- The behaviour of standard operators and built-in functions in python can be redefined for a new class using ***specially named methods :***

  Examples:

  + operator is overloaded by implementing a method named `__add__`

- Non-operator overload

  `str(foo)` is overloaded for an object by implementing a method `foo.__str__().`

  An user-defined class 'foo' can be treated as bool variable by implementing `foo.__bool__()` method.

- If a particular special method is not implemented in a user-defined class, the standard syntax relies upon that method will raise an exception.

  E.g. : a+b will raise an error if `__add__` is not defined.

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)
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, ...)	
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__()	

Example: Operator overloading for a vector class

```python
class Vector:
  '''
  Represent a vector in a multidimensional space.

  '''
  def __init__(self, d):
    '''Create d-dimensional vector of zeros.'''
    self._coords = [0] * d

  def __len__(self):
    '''Return the dimension of the vector.'''
    return len(self._coords)

  def __getitem__(self, j):
    '''Return jth coordinate of vector.'''
    return self._coords[j]

  def __setitem__(self, j, val):
    '''Set jth coordinate of vector to given value.'''
    self._coords[j] = val

  def __add__(self, other):
    '''Return sum of two vectors.'''
    if len(self) != len(other): # relies on __len__ method
      raise ValueError( 'dimensions must agree' )
    result = Vector(len(self)) # start with vector of zeros
    for j in range(len(self)):
      result[j] = self[j] + other[j]
    return result

  def __eq__(self, other):
    '''Return True if vector has same coordinates as other.'''
    return self._coords == other._coords

  def __ne__(self, other):
    '''Return True if vector differs from other.'''
    return not self == other # rely on existing __eq__ definition

  def __str__(self):
    '''Produce string representation of vector.'''
    return '<' + str(self._coords)[1:-1] + '>' # adapt list representation
```

- ***Implied Methods:*** There are some operators that have default definitions provided by Python, in the absence of special methods, and there are some operators whose definitions are derived from others:

  For example, the __bool__ method, which supports the syntax if foo:, has default semantics so that every object other than None is evaluated as True.

  However, if __len__ method is defined, then bool(foo) is interpreted by default to be True for instances with nonzero length

```python
########

a = Vector(5)
b = Vector(5)
c = Vector(5)
d = Vector(0)

print( 'a dimension:', len(a))
print( 'b dimension:', len(b))

a[2] = 3
b[3] = 2

c = a + b  # overload + operator

print('c:', c)  # overloaded str operator

total = 0;
for entry in c:  # implicit iteration via __len__ and __getitem__
  total += entry

if bool(d):   # Uses implied meaning of bool function
  print('The vector has non zero length')
else:
  print('The vector has zero length')
```

```
a dimension: 5
b dimension: 5
c: <0, 0, 3, 2, 0>
The vector has zero length
```

Iterator for a Class

Example: Creating a class iterator using generator syntax

- An ***iterator*** for a collection provides one key behavior - It supports a special method named **`__next__`** that returns the next element of the collection, if any, or raises a **`StopIteration`** exception to indicate that there are no further elements.

- There are two ways to implement an iterator
  - Using the ***generator*** syntax: `__next__` and `__iter__`
  - By defining `__len__` and `__getitem__` methods for the user-defined class.

```
class SequenceIterator:
  '''An iterator for any of Python s sequence types.'''

  def __init__(self, sequence):
    '''Create an iterator for the given sequence.'''
    self._seq = sequence # keep a reference to the underlying data
    self._k = -1 # will increment to 0 on first call to next

  def __next__(self):
    '''Return the next element, or else raise StopIteration error.'''
    self._k += 1 # advance to next index
    if self._k < len(self._seq):
      return(self._seq[self._k]) # return the data element
    else:
      raise StopIteration() # there are no more elements

  def __iter__(self):
    '''By convention, an iterator must return itself as an iterator.'''
    return self

################

data = [7, 8, 9, 2, 3, 5]

for i in SequenceIterator(data):
  print(i)
```

```
7
8
9
2
3
5
```

Creating Class iterator using `__len__` and `__getitem__` function

- Re-implementation of Python’s range() function.
- It is possible to execute a for loop over a range.

```python
class Range:
  '''A class that mimics the built-in range class.'''

  def __init__(self, start, stop=None, step=1):
    '''
    Initialize a Range instance.
    Semantics is similar to built-in range class.
    '''
    if step == 0:
      raise ValueError( 'step cannot be 0' )

    if stop is None: # special case of range(n)
      start, stop = 0, start # should be treated as if range(0,n)

    # calculate the effective length once
    self._length = max(0, (stop - start + step - 1) // step)

    # need knowledge of start and step (but not stop) to support getitem
    self._start = start
    self._step = step

  def __len__(self):
    '''Return number of entries in the range.'''
    return self._length

  def __getitem__(self, k):
    '''Return entry at index k (using standard interpretation if negative).'''
    if k < 0:
      k += len(self) # attempt to convert negative index

    if not 0 <= k < self._length:
      raise IndexError( 'index out of range' )

    return self._start + k * self._step


##############

r = Range(8, 140, 5)
print('Length of r = ',len(r))
print('Sixteenth element of r =', r[15])

for i in r:
  print(i, end=' ')
print()

for i in range(0,27):
  print(r[i], end=' ')
print()

for i in Range(8,140, 5):
  print(i, end=' ')
```

```
Length of r =  27
Sixteenth element of r = 83
8 13 18 23 28 33 38 43 48 53 58 63 68 73 78 83 88 93 98 103 108 113 118 123 128 133 138
8 13 18 23 28 33 38 43 48 53 58 63 68 73 78 83 88 93 98 103 108 113 118 123 128 133 138
8 13 18 23 28 33 38 43 48 53 58 63 68 73 78 83 88 93 98 103 108 113 118 123 128 133 138
```

Inheritance

- A natural way to organize various structural components of a software package is in a ***hierarchical*** fashion.

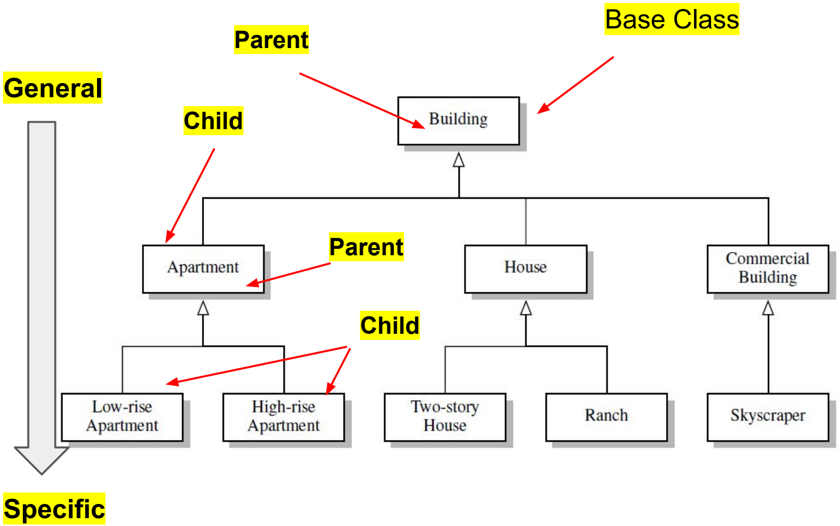
- Similar abstract functions are grouped together in a level-by-level manner.

- The abstraction goes from specific to more general as one traverses up the hierarchy.

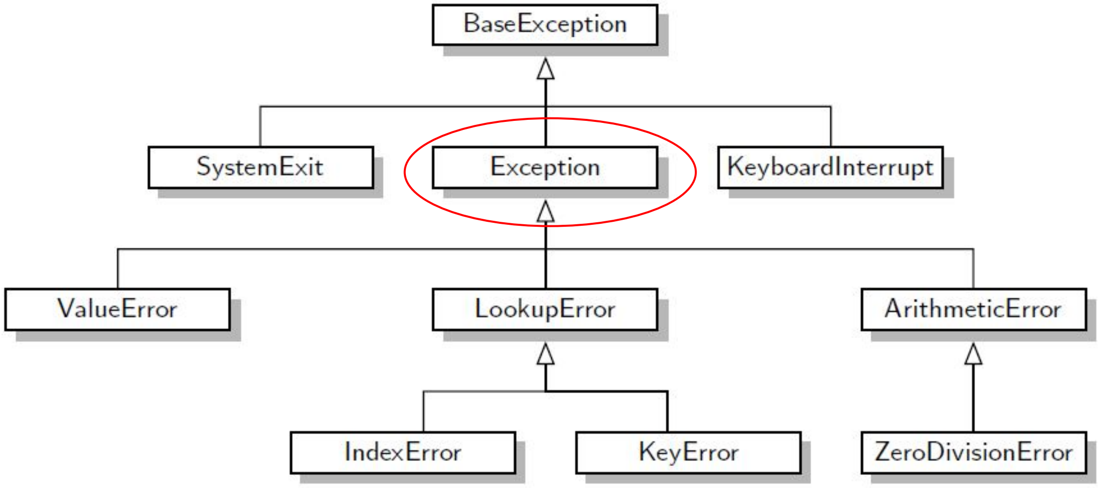
- ***Inheritance*** allows a new class called (***subclass*** or ***child class***) to be defined based upon an existing class (***base class***, ***parent class*** or ***superclass***) as the starting point.

- A subclass may ***specialize*** an existing behaviour by providing a new implementation that ***overrides*** an existing method.

- A subclass may also ***extend*** its superclass by Providing brand new methods.



Example: Python's hierarchy of Exception Types



A portion of Python's hierarchy of exception types.

Example: Credit Card Class

```python
# Credit Card Example

class CreditCard:

  def __init__(self, customer, bank, acnt, limit):


    self._customer = customer
    self._bank = bank
    self._account = acnt
    self._limit = limit
    self._balance = 0


  def get_customer(self):
    return self._customer

  def get_bank(self):
    return self._bank

  def get_account(self):
    return self._account

  def get_limit(self):
    return self._limit

  def get_balance(self):
    return self._balance


  def charge(self, price):
    '''
    Charge given price to the card, assuming sufficient credit limit.
    Return True if charge was process; fale if charge was denied
    '''

    if price + self._balance > self._limit:
      return False
    else:
      self._balance += price
      return True

  def make_payment(self, amount):
    '''
    Process customer payment that reduces the balance
    '''
    self._balance -= amount
```

```python
# Test Module


if __name__ == '__main__':
  wallet = []
  wallet.append(CreditCard('John Bowman' , 'California Savings' ,
                           '5391 0375 9387 5309' , 2500) )
  wallet.append(CreditCard('John Bowman' , 'California Federal' ,
                           '3485 0399 3395 1954' , 3500) )
  wallet.append(CreditCard('John Bowman' , 'California Finance' ,
                           '5391 0375 9387 5309' , 5000) )

  for val in range(1,17):
    wallet[0].charge(val)
    wallet[1].charge(2*val)
    wallet[2].charge(3*val)

  for c in range(3):
    print('Customer =', wallet[c].get_customer())
    print('Bank = ', wallet[c].get_bank())
    print('Account = ', wallet[c].get_account())
    print('Limit = ', wallet[c].get_limit())
    print('Balance =', wallet[c].get_balance())
    while wallet[c].get_balance() > 100:
      wallet[c].make_payment(100)
      print('New Balance = ', wallet[c].get_balance())
    print('----------------------')
```

```
Customer = John Bowman
Bank =  California Savings
Account =  5391 0375 9387 5309
Limit =  2500
Balance = 136
New Balance =  36
----------------------
Customer = John Bowman
Bank =  California Federal
Account =  3485 0399 3395 1954
Limit =  3500
Balance = 272
New Balance =  172
New Balance =  72
----------------------
Customer = John Bowman
Bank =  California Finance
Account =  5391 0375 9387 5309
Limit =  5000
Balance = 408
New Balance =  308
New Balance =  208
New Balance =  108
New Balance =  8
----------------------
```

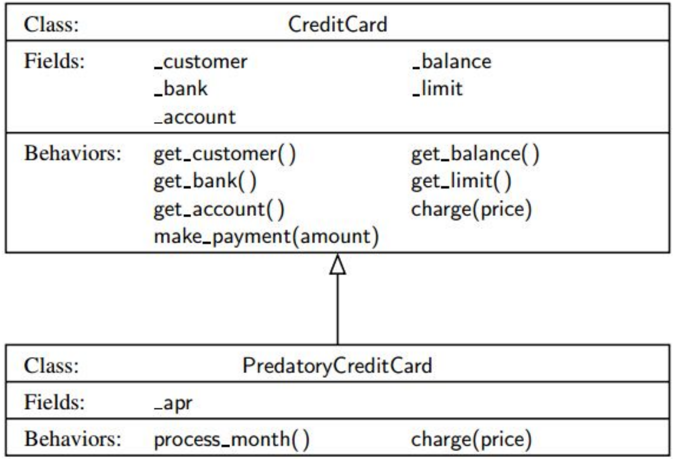
It has following functionality

- Create new account for customer
- Allows to fetch customer related data
- Expenses could be charged to the card
- User can make payment to maintain the card balance.
- Expenses can not exceed the card limit

Example: Inherited Class - PredatoryCreditCard

```python
# Inheritance example - Extension of credit card program
# Run the Credit card example above.

class PredatoryCreditCard(CreditCard):
  ''' An extension to CreditCard that compounds interest and fees.'''

  def __init__(self, customer, bank, acnt, limit, apr):
    '''
    Create a new predatory credit card instance.
    The initial balance is zero.
    customer: the name of the customer (e.g., John Bowman )
    bank: the name of the bank (e.g., California Savings )
    acnt: the acount identifier (e.g., 5391 0375 9387 5309 )
    limit: credit limit (measured in dollars)
    apr: annual percentage rate (e.g., 0.0825 for 8.25% APR)
    '''
    super().__init__(customer, bank, acnt, limit)  # call super constructor
    self._apr = apr


  def charge(self, price): # modified inherited behaviour
    '''
    Charge given price to the card, assuming sufficient credit limit.
    Return True if charge was processed.
    Return False and assess 5 fee if charge is denied.
    '''

    success = super().charge(price) # call inherited method
    if not success:
      self._balance += 5  # assess penalty
    return success


  def process_month(self):  # New behaviour in the child class
    '''
    Assess monthly interest on outstanding balance
    '''
    if self._balance > 0:
      # if positive balanec, convert APR to monthly multiplicative factor
      monthly_factor = pow(1 + self._apr, 1/12)
      self._balance *= monthly_factor


##################

pcc = PredatoryCreditCard('John Doe', '1st Bank', '5391 0375 9387 5309',
                          1000, 0.0825)

pcc.charge(2000)
print('Available Balance:', pcc.get_balance())

pcc.process_month()
print('Available Balance after processing:',pcc.get_balance())
```

```
Available Balance: 5
Available Balance after processing: 5.03313983402184
```



This child class

- Inherits the constructor of parent class to create new customers
- Modifies the behaviour of charging the card by levying $5 if charge is denied.
- Modifies the card payment function by charging interest on overdue balance.

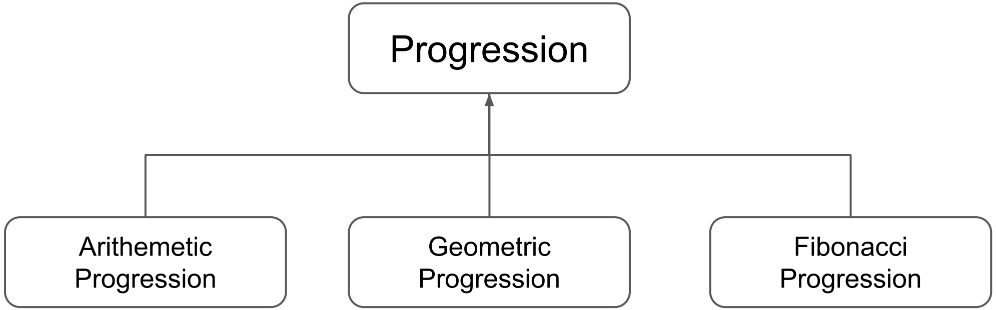
Another Example of Inheritance: Progression

```python
# Progression Class

class Progression:
  '''
  Iterator producing a generic progression.
  Default iterator produces the whole numbers 0, 1, 2, ...
  '''

  def __init__(self, start=0):
    '''Initialize current to the first value of the progression.'''

    self._current = start

  def _advance(self):
    '''
    Update self._current to a new value.
    This should be overridden by a subclass to customize progression.
    By convention, if current is set to None, this designates the
    end of a finite progression.
    '''

    self._current += 1

  def __next__(self):
    '''Return the next element, or else raise StopIteration error.'''

    if self._current is None: # our convention to end a progression
      raise StopIteration()
    else:
      answer = self._current # record current value to return
      self._advance( ) # advance to prepare for next time
      return answer # return the answer

  def __iter__(self):
    '''By convention, an iterator must return itself as an iterator.'''
    return self

  def print_progression(self, n):
    '''Print next n values of the progression.'''
    print(' '.join(str(next(self)) for j in range(n)))  #next() is overloaded here.

########################

seq = Progression()
seq.print_progression(10)
```

```
0 1 2 3 4 5 6 7 8 9
```

Base Class

- Produces a general sequence
- It uses generator syntax (`__next__`, `__iter__`) to provide iteration capabilities.
- Create 3 new child classes to extend the capability of this base class



```python
# Arithmetic Progression
# Execute the code for "Progression Class before executing this cell"

class ArithmeticProgression(Progression): # inherit from Progression
  '''Iterator producing an arithmetic progression.'''

  def __init__(self, increment=2, start=0):
    '''
    Create a new arithmetic progression.
    increment: the fixed constant to add to each term (default 1)
    start: the first term of the progression (default 0)
    '''
    super().__init__(start) # initialize base class
    self._increment = increment

  def _advance(self): # override inherited version
    '''Update current value by adding the fixed increment.'''
    self._current += self._increment


#################

a = ArithmeticProgression()
a.print_progression(10)
```

```
0 2 4 6 8 10 12 14 16 18
```

Arithmetic Progression

Geometric Progression

```python
# Geometric Progression
# Execute the code for "Progression Class" before executing this cell


class GeometricProgression(Progression): # inherit from Progression
  '''Iterator producing a geometric progression.'''

  def __init__(self, base=2, start=1):
    '''
    Create a new geometric progression.
    base: the fixed constant multiplied to each term (default 2)
    start: the first term of the progression (default 1)
    '''

    super().__init__(start)
    self._base = base

  def _advance(self): # override inherited version
    '''Update current value by multiplying it by the base value.'''
    self._current *= self._base

 ####################

b = GeometricProgression()
b.print_progression(10)

```

```
1 2 4 8 16 32 64 128 256 512
```

```python
# Fibonacci Progression
# Run the Progression class above first

class FibonacciProgression(Progression):
  '''Iterator producing a generalized Fibonacci progression.'''

  def __init__(self, first=0, second=1):
    '''Create a new fibonacci progression.

    first the first term of the progression (default 0)
    second the second term of the progression (default 1)
    '''
    super().__init__(first) # start progression at first
    self._prev = second - first # fictitious value preceding the first

  def _advance(self):
    '''Update current value by taking sum of previous two.'''
    self._prev, self._current = self._current, self._prev + self._current

######################

c = FibonacciProgression()
c.print_progression(10)

FibonacciProgression(4,6).print_progression(10)
```

```
0 1 1 2 3 5 8 13 21 34
4 6 10 16 26 42 68 110 178 288
```

Fibonacci Progression

- Each child class modifies the base class constructor `__init__()`
- Modifies the `_advance()` method

Abstract Classes

- An abstract class can be considered as a blueprint or template for other classes.
- An abstract class is a class that contains one or more abstract methods.
- An abstract method is a method that has declaration but no implementation.
- Abstract classes can not be instantiated. It needs subclasses (child classes) to provide implementation.
- Abstract classes are required for providing "Abstraction" or a simplified interface (API) while hiding the underlying implementation.
- Python provides abstract classes by declaring abstract base class (ABC) which could be inherited by other child classes.

we cannot create a object of abstract class but we can create children classes

```python
# Python program showing
# abstract base class work

from abc import ABC, abstractmethod
class Animal(ABC):   # base class

  @abstractmethod
  def move(self): # Abstract method
    pass

class Human(Animal): # Child class 1

  def move(self):
    print("I can walk and run")

class Snake(Animal): # Child Class 2

  def move(self):
    print("I can crawl")

class Dog(Animal): # Child class 3

  def move(self):
    print("I can bark")

class Lion(Animal):  # Child class 4

  def move(self):
    print("I can roar")

# Driver code
R = Human()
R.move()

K = Snake()
K.move()

R = Dog()
R.move()

K = Lion()
K.move()

L = Animal() # Causes error
L.move()
```

```
I can walk and run
I can crawl
I can bark
I can roar
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-26-94fbb05ae137> in <module>()
     40 K.move()
     41
---> 42 L = Animal() # Causes error
     43 L.move()
     44

TypeError: Can't instantiate abstract class Animal with abstract methods move
```

@abstractmethod makes it compulsory to redefine method in child class

Example 1

```python
import abc
from abc import ABC, abstractmethod
class AbstractClassExample(ABC):
  def __init__(self, value):
    self.value = value
    super().__init__()

  @abstractmethod
  def do_something(self):
    pass

class DoAdd42(AbstractClassExample):

  def do_something(self):
    return self.value + 42



X = DoAdd42(4)
print(X.do_something())
```

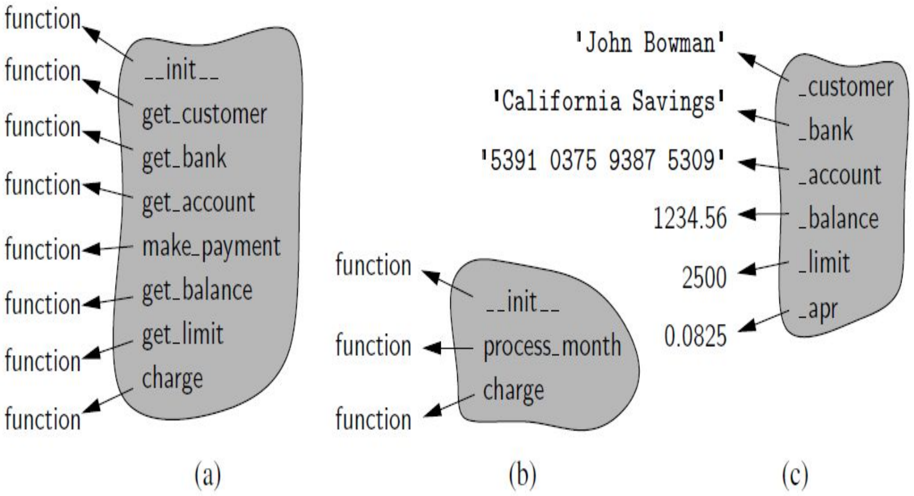
```
46
```

Example 2

- X inherits base class constructor for initialization
- X re-defines abstract method in base class

Namespaces & Object-Orientation

- A ***class namespace*** includes all declarations that are made directly within the body of the class definition.
-



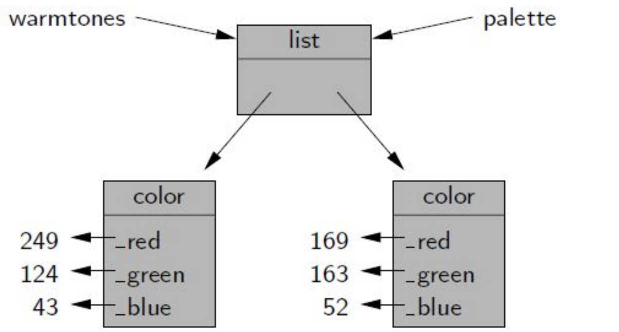
Class namespace of CreditCard

Class namespace of PredatoryCreditCard

Instance namespace for a PredatoryCreditCard object

Shallow & Deep Copying

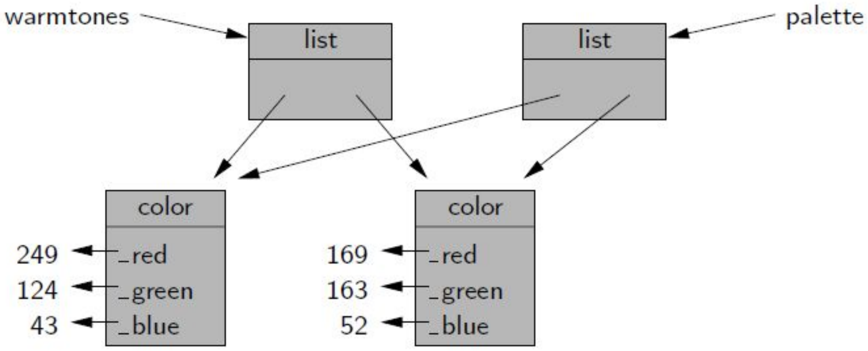
- Different methods for copying data
  - Aliases
  - Shallow copy
  - Deep Copy



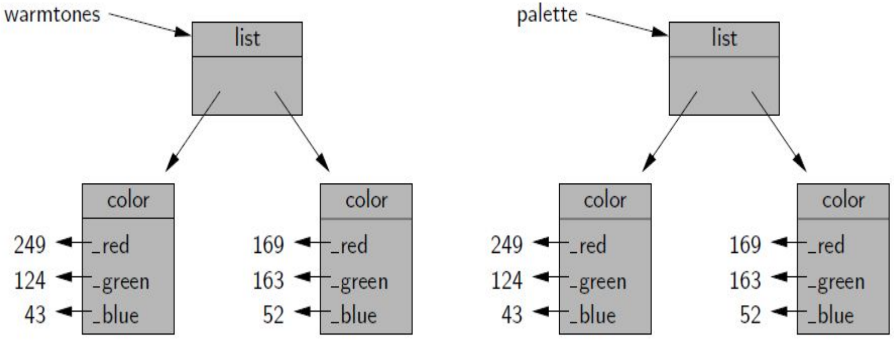
Two aliases for the same list of colors.

palette = warmtones

If pallette is changed, warmtone changes as well. They share the same memory location.



A shallow copy of a list of colors.



A deep copy of a list of colors.

```
palette = list(warmtones)
```

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

- We can add and remove elements from pallette without affecting warmtones.
- But we can not edit a color instance from the pallette list. It will affect the warmtone colors.
- Although they are distinct lists, there remains indirect aliasing, for example, pallette[0] and warmtones[0] as aliases for the same color instance.

- In deep copy, the new copy references its own copies of those referenced by the original version.
- It creates separate memory location for two copies.
- Both copies could be modified independently without affecting each other.

Summary

In this module, we covered the following:

- What is OOPs and why is it needed?
- How to create classes and objects?
- How to overload standard operators and functions for user-defined classes?
- How to structure programs through inheritance?
- How to create and use Abstract Classes?
- Difference between shallow and deep copying.