

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 `'__'`

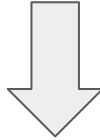
```
1 class Person:
2     def __init__(self, name=None, age=None, gender=None):
3         self._name = name # protected or nonpublic
4         self._age = age
5         self.__gender = gender # private
6
7     def get_attrib(self):
8         print("Hello my name is ", self._name)
9         print("My age is", self._age)
10        print("My Gender is", self.__gender)
11
12    def set_attrib(self, name, age, gender):
13        self._name=name
14        self._age = age
15        self.__gender = gender
16
17    p1 = Person("John", 36, "Male")
18    p1.get_attrib()
19    p1.set_attrib("Harry", 25, "Male")
20    p1.get_attrib()
21
22    p2 = Person()
23    p2.get_attrib()
24
25    p2.__gender = "Female" # still accessible
26    p2._name = "Sally" # Still accessible
27    p2._age = 23 # Still accessible
28
29    p2.get_attrib()
30
31    print("Gender of P2 is", p2.__gender)]
32
```

```
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 ??
```

```
1 # Operator overloading  
2 print(2+3) # addition of numbers  
3 print([2,3] + [4,5])  
4 a = [[2,3],[4,5]]  
5 b = [[7,8],[10,11]]  
6 c = a + b # extending an array  
7 print(c)  
8 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
<code>a + b</code>	<code>a.__add__(b);</code> alternatively <code>b.__radd__(a)</code>
<code>a - b</code>	<code>a.__sub__(b);</code> alternatively <code>b.__rsub__(a)</code>
<code>a * b</code>	<code>a.__mul__(b);</code> alternatively <code>b.__rmul__(a)</code>
<code>a / b</code>	<code>a.__truediv__(b);</code> alternatively <code>b.__rtruediv__(a)</code>
<code>a // b</code>	<code>a.__floordiv__(b);</code> alternatively <code>b.__rfloordiv__(a)</code>
<code>a % b</code>	<code>a.__mod__(b);</code> alternatively <code>b.__rmod__(a)</code>
<code>a ** b</code>	<code>a.__pow__(b);</code> alternatively <code>b.__rpow__(a)</code>
<code>a << b</code>	<code>a.__lshift__(b);</code> alternatively <code>b.__rlshift__(a)</code>
<code>a >> b</code>	<code>a.__rshift__(b);</code> alternatively <code>b.__rrshift__(a)</code>
<code>a & b</code>	<code>a.__and__(b);</code> alternatively <code>b.__rand__(a)</code>
<code>a ^ b</code>	<code>a.__xor__(b);</code> alternatively <code>b.__rxor__(a)</code>
<code>a b</code>	<code>a.__or__(b);</code> alternatively <code>b.__ror__(a)</code>
<code>a += b</code>	<code>a.__iadd__(b)</code>
<code>a -= b</code>	<code>a.__isub__(b)</code>
<code>a *= b</code>	<code>a.__imul__(b)</code>
...	...
<code>+a</code>	<code>a.__pos__()</code>
<code>-a</code>	<code>a.__neg__()</code>
<code>~a</code>	<code>a.__invert__()</code>
<code>abs(a)</code>	<code>a.__abs__()</code>
<code>a < b</code>	<code>a.__lt__(b)</code>
<code>a <= b</code>	<code>a.__le__(b)</code>
<code>a > b</code>	<code>a.__gt__(b)</code>
<code>a >= b</code>	<code>a.__ge__(b)</code>
<code>a == b</code>	<code>a.__eq__(b)</code>
<code>a != b</code>	<code>a.__ne__(b)</code>
<code>v in a</code>	<code>a.__contains__(v)</code>
<code>a[k]</code>	<code>a.__getitem__(k)</code>
<code>a[k] = v</code>	<code>a.__setitem__(k,v)</code>
<code>del a[k]</code>	<code>a.__delitem__(k)</code>
<code>a(arg1, arg2, ...)</code>	<code>a.__call__(arg1, arg2, ...)</code>
<code>len(a)</code>	<code>a.__len__()</code>
<code>hash(a)</code>	<code>a.__hash__()</code>
<code>iter(a)</code>	<code>a.__iter__()</code>
<code>next(a)</code>	<code>a.__next__()</code>
<code>bool(a)</code>	<code>a.__bool__()</code>
<code>float(a)</code>	<code>a.__float__()</code>
<code>int(a)</code>	<code>a.__int__()</code>
<code>repr(a)</code>	<code>a.__repr__()</code>
<code>reversed(a)</code>	<code>a.__reversed__()</code>
<code>str(a)</code>	<code>a.__str__()</code>

Example: Operator overloading for a vector class

```
1 class Vector:
2     """
3     Represent a vector in a multidimensional space.
4
5     """
6     def __init__(self, d):
7         """Create d-dimensional vector of zeros."""
8         self._coords = [0] * d
9
10    def __len__(self):
11        """Return the dimension of the vector."""
12        return len(self._coords)
13
14    def __getitem__(self, j):
15        """Return jth coordinate of vector."""
16        return self._coords[j]
17
18    def __setitem__(self, j, val):
19        """Set jth coordinate of vector to given value."""
20        self._coords[j] = val
21
22    def __add__(self, other):
23        """Return sum of two vectors."""
24        if len(self) != len(other): # relies on __len__ method
25            raise ValueError('dimensions must agree')
26        result = Vector(len(self)) # start with vector of zeros
27        for j in range(len(self)):
28            result[j] = self[j] + other[j]
29        return result
30
31    def __eq__(self, other):
32        """Return True if vector has same coordinates as other."""
33        return self._coords == other._coords
34
35    def __ne__(self, other):
36        """Return True if vector differs from other."""
37        return not self == other # rely on existing __eq__ definition
38
39    def __str__(self):
40        """Produce string representation of vector."""
41        return '<' + str(self._coords)[1:-1] + '>' # adapt list representation
42
```

- **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

```
43 #####
44
45 a = Vector(5)
46 b = Vector(5)
47 c = Vector(5)
48 d = Vector(0)
49
50 print('a dimension:', len(a))
51 print('b dimension:', len(b))
52
53 a[2] = 3
54 b[3] = 2
55
56 c = a + b # overload + operator
57
58 print('c:', c) # overloaded str operator
59
60 total = 0;
61 for entry in c: # implicit iteration via __len__ and __getitem__
62     total += entry
63
64 if bool(d): # Uses implied meaning of bool function
65     print('The vector has non zero length')
66 else:
67     print('The vector has zero length')
68
```

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

Iterator for a Class

- 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.

Example: Creating a class iterator using generator syntax

```
1 class SequenceIterator:
2     '''An iterator for any of Python's sequence types.'''
3
4     def __init__(self, sequence):
5         '''Create an iterator for the given sequence.'''
6         self._seq = sequence # keep a reference to the underlying data
7         self._k = -1 # will increment to 0 on first call to next
8
9     def __next__(self):
10        '''Return the next element, or else raise StopIteration error.'''
11        self._k += 1 # advance to next index
12        if self._k < len(self._seq):
13            return(self._seq[self._k]) # return the data element
14        else:
15            raise StopIteration() # there are no more elements
16
17    def __iter__(self):
18        '''By convention, an iterator must return itself as an iterator.'''
19        return self
20
21    #####
22
23    data = [7, 8, 9, 2, 3, 5]
24
25    for i in SequenceIterator(data):
26        print(i)
27
```

7
8
9
2
3
5

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

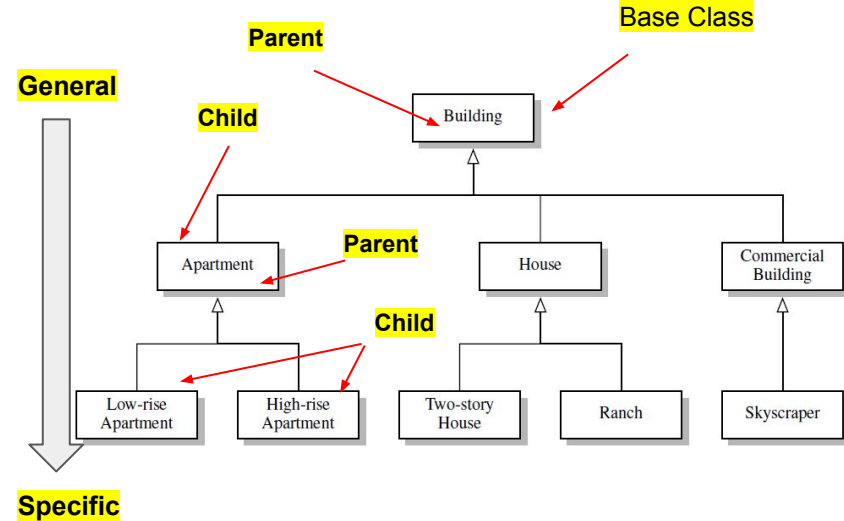
- Re-implementation of Python's `range()` function.
- It is possible to execute a for loop over a range.

```
1 class Range:
2     '''A class that mimics the built-in range class.'''
3
4     def __init__(self, start, stop=None, step=1):
5         ..
6         Initialize a Range instance.
7         Semantics is similar to built-in range class.
8         ..
9         if step == 0:
10             raise ValueError( 'step cannot be 0' )
11
12         if stop is None: # special case of range(n)
13             start, stop = 0, start # should be treated as if range(0,n)
14
15         # calculate the effective length once
16         self._length = max(0, (stop - start + step - 1) // step)
17
18         # need knowledge of start and step (but not stop) to support getitem
19         self._start = start
20         self._step = step
21
22     def __len__(self):
23         '''Return number of entries in the range.'''
24         return self._length
25
26     def __getitem__(self, k):
27         '''Return entry at index k (using standard interpretation if negative).'''
28         if k < 0:
29             k += len(self) # attempt to convert negative index
30
31         if not 0 <= k < self._length:
32             raise IndexError( 'index out of range' )
33
34         return self._start + k * self._step
35
36 #####
37
38 r = Range(8, 140, 5)
39
40 print('Length of r = ', len(r))
41 print('Sixteenth element of r =', r[15])
42
43 for i in r:
44     print(i, end=' ')
45     print()
46
47 for i in range(0, 27):
48     print(r[i], end='')
49     print()
50
51 for i in Range(8, 140, 5):
52     print(i, end=' ')
53
54
```

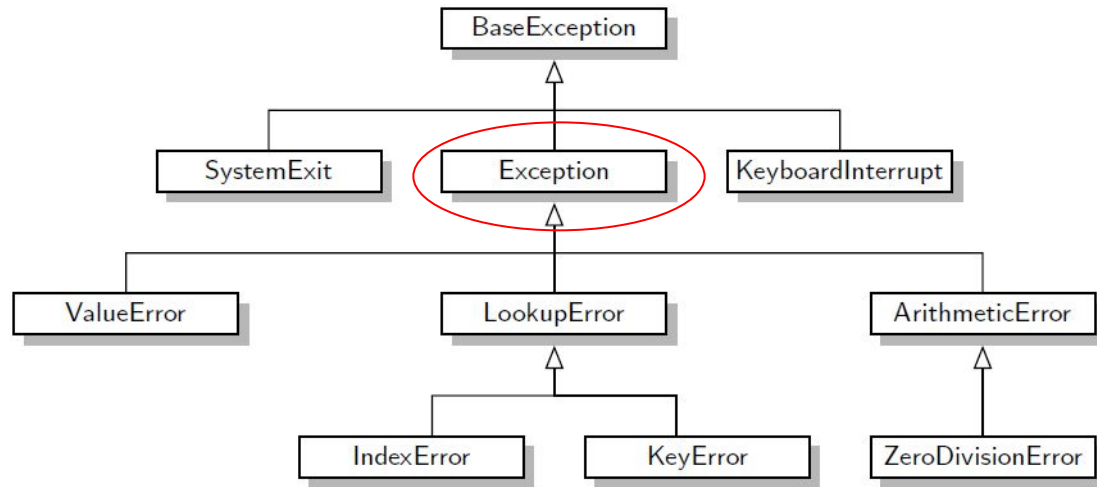
```
Length of r = 27
Sixteenth element of r = 83
8 13 18 23 28 33 38 43 48 53 58 63 68 73 78 83 88 93 98 103 108 113 118 123 128 133 138
8 13 18 23 28 33 38 43 48 53 58 63 68 73 78 83 88 93 98 103 108 113 118 123 128 133 138
8 13 18 23 28 33 38 43 48 53 58 63 68 73 78 83 88 93 98 103 108 113 118 123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

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

```
1 # Credit Card Example
2
3 class CreditCard:
4
5     def __init__(self, customer, bank, acct, limit):
6
7         self._customer = customer
8         self._bank = bank
9         self._account = acct
10        self._limit = limit
11        self._balance = 0
12
13
14
15    def get_customer(self):
16        return self._customer
17
18    def get_bank(self):
19        return self._bank
20
21    def get_account(self):
22        return self._account
23
24    def get_limit(self):
25        return self._limit
26
27    def get_balance(self):
28        return self._balance
29
30
31    def charge(self, price):
32        """
33        Charge given price to the card, assuming sufficient credit limit.
34        Return True if charge was process; false if charge was denied
35        """
36
37        if price + self._balance > self._limit:
38            return False
39        else:
40            self._balance += price
41            return True
42
43    def make_payment(self, amount):
44        """
45        Process customer payment that reduces the balance
46        """
47        self._balance -= amount
48
```

```
1 # Test Module
2
3 if __name__ == '__main__':
4     wallet = []
5     wallet.append(CreditCard('John Bowman', 'California Savings',
6                             '5391 0375 9387 5309', 2500) )
7     wallet.append(CreditCard('John Bowman', 'California Federal',
8                             '3485 0399 3395 1954', 3500) )
9     wallet.append(CreditCard('John Bowman', 'California Finance',
10                             '5391 0375 9387 5309', 5000) )
11
12     for val in range(1,17):
13         wallet[0].charge(val)
14         wallet[1].charge(2*val)
15         wallet[2].charge(3*val)
16
17     for c in range(3):
18         print('Customer =', wallet[c].get_customer())
19         print('Bank = ', wallet[c].get_bank())
20         print('Account = ', wallet[c].get_account())
21         print('Limit = ', wallet[c].get_limit())
22         print('Balance = ', wallet[c].get_balance())
23         while wallet[c].get_balance() > 100:
24             wallet[c].make_payment(100)
25             print('New Balance = ', wallet[c].get_balance())
26         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

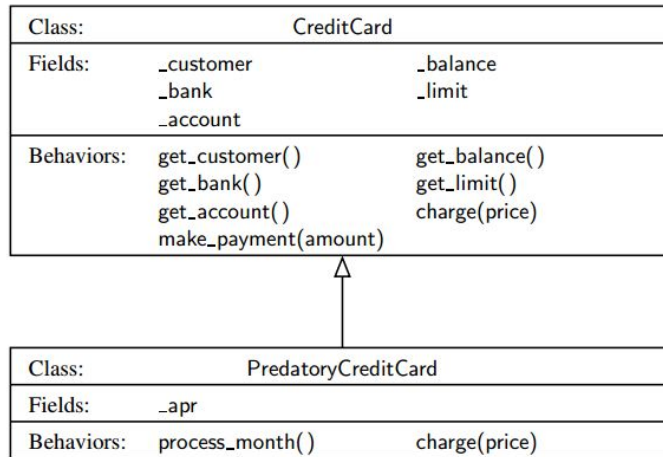
- Create new account for customer
- Allows to fetch customer related data
- Expenses could be charged to the card
- User can make payment to maintain the card balance.
- Expenses can not exceed the card limit

Example: Inherited Class - PredatoryCreditCard

```
1 # Inheritance example - Extension of credit card program
2 # Run the Credit card example above.
3
4 class PredatoryCreditCard(CreditCard):
5     ''' An extension to CreditCard that compounds interest and fees.'''
6
7     def __init__(self, customer, bank, acct, limit, apr):
8         '''
9         Create a new predatory credit card instance.
10        The initial balance is zero.
11        customer: the name of the customer (e.g., John Bowman )
12        bank: the name of the bank (e.g., California Savings )
13        acct: the account identifier (e.g., 5391 0375 9387 5309 )
14        limit: credit limit (measured in dollars)
15        apr: annual percentage rate (e.g., 0.0825 for 8.25% APR)
16        '''
17        super().__init__(customer, bank, acct, limit) # call super constructor
18        self._apr = apr
19
20
21     def charge(self, price): # modified inherited behaviour
22         '''
23         Charge given price to the card, assuming sufficient credit limit.
24         Return True if charge was processed.
25         Return False and assess 5 fee if charge is denied.
26         '''
27
28         success = super().charge(price) # call inherited method
29         if not success:
30             self._balance += 5 # assess penalty
31         return success
32
33
34     def process_month(self): # New behaviour in the child class
35         '''
36         Assess monthly interest on outstanding balance
37         '''
38         if self._balance > 0:
39             # if positive balance, convert APR to monthly multiplicative factor
40             monthly_factor = pow(1 + self._apr, 1/12)
41             self._balance *= monthly_factor
42
43
44     #####
45
46 pcc = PredatoryCreditCard('John Doe', '1st Bank', '5391 0375 9387 5309',
47                           1000, 0.0825)
48
49 pcc.charge(2000)
50 print('Available Balance:', pcc.get_balance())
51
52 pcc.process_month()
53 print('Available Balance after processing:', pcc.get_balance())
54
55
```

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.

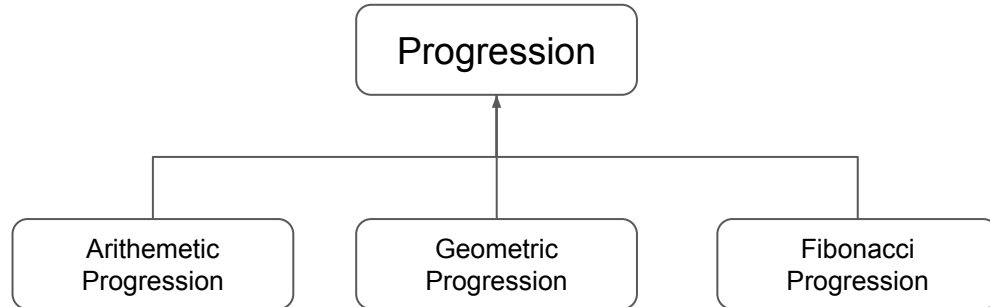
Another Example of Inheritance: Progression

```
1 # Progression Class
2
3 class Progression:
4     """
5     Iterator producing a generic progression.
6     Default iterator produces the whole numbers 0, 1, 2, ...
7     """
8
9     def __init__(self, start=0):
10         """Initialize current to the first value of the progression."""
11
12         self._current = start
13
14     def _advance(self):
15         """
16         Update self._current to a new value.
17         This should be overridden by a subclass to customize progression.
18         By convention, if current is set to None, this designates the
19         end of a finite progression.
20         """
21
22         self._current += 1
23
24     def __next__(self):
25         """Return the next element, or else raise StopIteration error."""
26
27         if self._current is None: # our convention to end a progression
28             raise StopIteration()
29         else:
30             answer = self._current # record current value to return
31             self._advance() # advance to prepare for next time
32             return answer # return the answer
33
34     def __iter__(self):
35         """By convention, an iterator must return itself as an iterator."""
36         return self
37
38     def print_progression(self, n):
39         """Print next n values of the progression."""
40         print(' '.join(str(next(self)) for j in range(n))) #next() is overloaded here.
41
42 #####
43
44 seq = Progression()
45 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




```

1 # Arithmetic Progression
2 # Execute the code for "Progression Class before executing this cell"
3
4 class ArithmeticProgression(Progression): # inherit from Progression
5     '''Iterator producing an arithmetic progression.'''
6
7     def __init__(self, increment=2, start=0):
8         '''
9         Create a new arithmetic progression.
10        increment: the fixed constant to add to each term (default 1)
11        start: the first term of the progression (default 0)
12        '''
13        super().__init__(start) # initialize base class
14        self._increment = increment
15
16    def _advance(self): # override inherited version
17        '''Update current value by adding the fixed increment.'''
18        self._current += self._increment
19
20
21 #####
22
23 a = ArithmeticProgression()
24 a.print_progression(10)

```

0 2 4 6 8 10 12 14 16 18

Arithmetic Progression

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

```

1 # Fibonacci Progression
2 # Run the Progression class above first
3
4 class FibonacciProgression(Progression):
5     '''Iterator producing a generalized Fibonacci progression.'''
6
7     def __init__(self, first=0, second=1):
8         '''Create a new fibonacci progression.
9
10        first the first term of the progression (default 0)
11        second the second term of the progression (default 1)
12        '''
13        super().__init__(first) # start progression at first
14        self._prev = second - first # fictitious value preceding the first
15
16    def _advance(self):
17        '''Update current value by taking sum of previous two.'''
18        self._prev, self._current = self._current, self._prev + self._current
19
20 #####
21
22 c = FibonacciProgression()
23 c.print_progression(10)
24
25 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

Geometric Progression

```

1 # Geometric Progression
2 # Execute the code for "Progression Class" before executing this cell
3
4 class GeometricProgression(Progression): # inherit from Progression
5     '''Iterator producing a geometric progression.'''
6
7     def __init__(self, base=2, start=1):
8         '''
9         Create a new geometric progression.
10        base: the fixed constant multiplied to each term (default 2)
11        start: the first term of the progression (default 1)
12        '''
13
14        super().__init__(start)
15        self._base = base
16
17    def _advance(self): # override inherited version
18        '''Update current value by multiplying it by the base value.'''
19        self._current *= self._base
20
21 #####
22
23 b = GeometricProgression()
24 b.print_progression(10)
25
26

```

1 2 4 8 16 32 64 128 256 512

Fibonacci Progression

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.

```

1 # Python program showing
2 # abstract base class work
3
4 from abc import ABC, abstractmethod
5 class Animal(ABC): # base class
6
7     @abstractmethod
8     def move(self): # Abstract method
9         pass
10
11 class Human(Animal): # Child class 1
12
13     def move(self):
14         print("I can walk and run")
15
16 class Snake(Animal): # Child Class 2
17
18     def move(self):
19         print("I can crawl")
20
21 class Dog(Animal): # Child class 3
22
23     def move(self):
24         print("I can bark")
25
26 class Lion(Animal): # Child class 4
27
28     def move(self):
29         print("I can roar")
30
31 # Driver code
32 R = Human()
33 R.move()
34
35 K = Snake()
36 K.move()
37
38 R = Dog()
39 R.move()
40
41 K = Lion()
42 K.move()
43
44 L = Animal() # Causes error
45 L.move()
46
47

```

Example 1

@abstractmethod makes it compulsory to redefine method in child class

```

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

```

1 import abc
2 from abc import ABC, abstractmethod
3 class AbstractClassExample(ABC):
4     def __init__(self, value):
5         self.value = value
6         super().__init__()
7
8     @abstractmethod
9     def do_something(self):
10         pass
11
12 class DoAdd42(AbstractClassExample):
13
14     def do_something(self):
15         return self.value + 42
16
17
18
19 X = DoAdd42(4)
20 print(X.do_something())

```

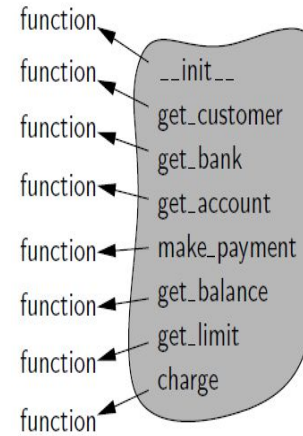
Example 2

46

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

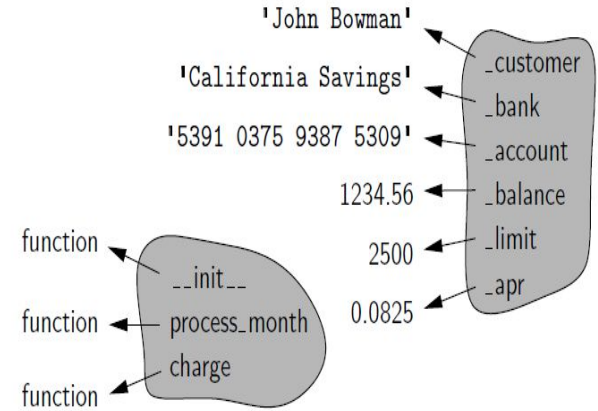
Namespaces & Object-Orientation

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



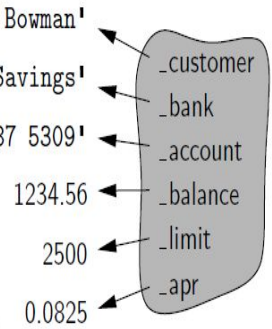
(a)

Class
namespace of
`CreditCard`



(b)

Class namespace of
`PredatoryCreditCard`

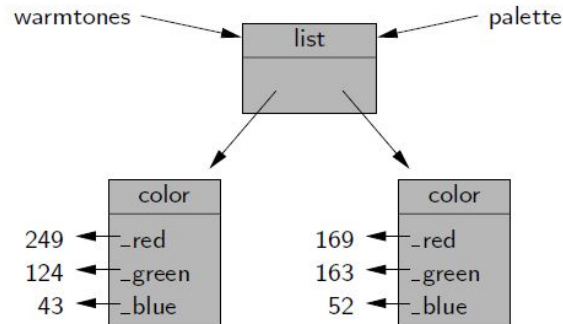


(c)

Instance namespace
for a
`PredatoryCreditCard`
object

Shallow & Deep Copying

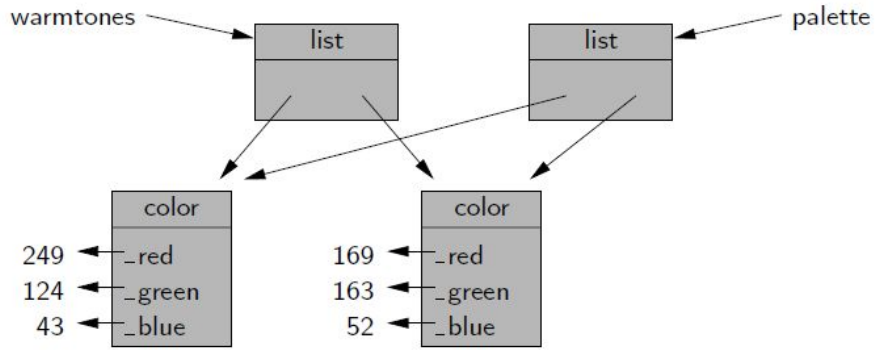
- Different methods for copying data
 - Aliases
 - Shallow copy
 - Deep Copy



Two aliases for the same list of colors.

`palette = warmtones`

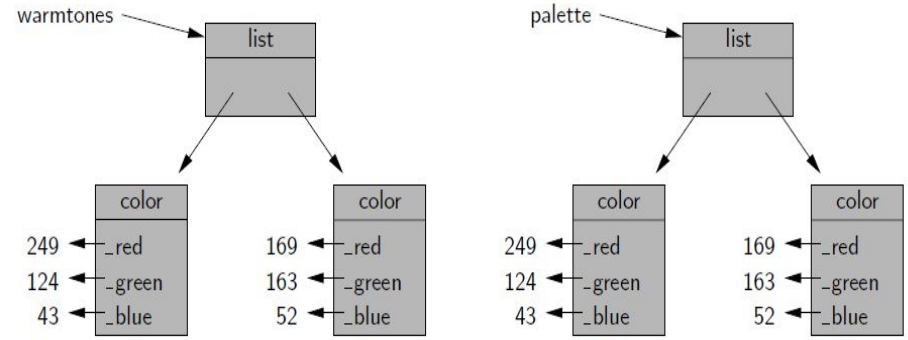
If `palette` is changed, `warmtone` changes as well. They share the same memory location.



A shallow copy of a list of colors.

`palette = list(warmtones)`

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



A deep copy of a list of colors.

`palette = copy.deepcopy(warmtones)`

- 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.