Python for Data Science

IISER-Kolkata ML4HEP Pre-school Lecture Series
May 13, 2025 - Lecture 3

Subir Sarkar, SINP
subir.sarkar@cern.ch

Module/Package

- A Python module is a file containing Python code
- A package is a collection of modules with a common purpose
- A project usually integrates different packages
- Python namespace is a combination of modules/packages

Module

Let us try to model a simple bank account

```
# bacc.py
def account():
    return {'balance': 0} # dictionary
def deposit(account, amount):
    account['balance'] += amount
    return account['balance']
def withdraw(account, amount):
    account['balance'] -= amount
    return account['balance']
```

http://anandology.com/python-practice-book/object_oriented_programming.html

A module in use

```
>>> from bacc import
account,deposit,withdraw
>>> a = account()
>>> b = account()
>>> deposit(a, 100)
100
>>> deposit(b, 50)
50
>>> withdraw(b, 10)
40
>>> withdraw(a, 10)
90
```

A Module w/ test code

```
def account():
    return {'balance': 0} # dictionary

def deposit(account, amount):
    account['balance'] += amount
    return account['balance']

def withdraw(account, amount):
    account['balance'] -= amount
    return account['balance']

def balance(account):
    return account['balance']


if __name__ == '__main__':
    a = account()
    b = account()
    deposit(a, 1000)
    deposit(b, 500)
    withdraw(b, 100)
    withdraw(a, 100)
    print('Balance -> A/C a:', balance(a), 'A/C b:', balance(b))
```

Object Oriented Programming - I

Object Oriented Programming

```
# account.py
class BankAccount:
    def __init__(self, amount=0): # c'tor
        self.__balance = amount
    def withdraw(self, amount): # methods
        self.__balance -= amount
        return self.__balance
    def deposit(self, amount):
        self.__balance += amount
        return self.__balance
```

Object Oriented Programming

```
>>> from account import BankAccount
>>> a = BankAccount(1000) # instances
>>> b = BankAccount()
>>> a.deposit(100) # -> deposit(a, 100)
100
>>> b.deposit(50)
50
>>> b.withdraw(10)
40
>>> a.withdraw(10)
90
```

The *self* parameter

- The first argument of every class instance method, including `__init__` and `__del__`, is an explicit reference to the current instance of the class
- by convention, this argument is always named *self, but it is not a keyword*

```
def __init__(self, amnt=0)
```

- In the `__init__` method, *`self`* refers to the object being created
- In other methods, *`self`* refers to the object
  - whenever an object calls its method, the object itself is passed as the first argument
- The constructor is defined with two arguments, while an object is created passing none!

```
a = BankAccount()  # 2nd parameter has default value
```

Operator Overloading

```
class Integer:
  def __init__(self, v=0):
    self._value = v
  def set(self, v):
    self._value = v
  def __add__(self, value):
    return self._value + value
  def __sub__(self, value):
    return self._value - value
  def __mul__(self, value):
    return self._value * value
  def __str__(self):
    return str(self._value)
  def __repr__(self):
    return 'Integer(%d)' % \
    self._value
```

Operator Overloading

```
class Integer:
  def __init__(self, v=0):
    self._value = v
  def set(self, v):
    self._value = v
  def __add__(self, value):
    return self._value + value
  def __sub__(self, value):
    return self._value - value
  def __mul__(self, value):
    return self._value * value
  def __str__(self):
    return str(self._value)
  def __repr__(self):
    return 'Integer(%d)' % \
    self._value
```

```
if __name__ == '__main__':
  d = Integer(10)
  print(type(d))

  print(d + 2)
  print(d - 2)
  print(d * 2)

  d = 10  # d.set(10)
  if d == 5:
    print("Yes!")
  else:
    print("no...")
```

Operator Overloading

Operator	Expression	Internally
Addition	p1 + p2	p1.__add__(p2)
Subtraction	p1 - p2	p1.__sub__(p2)
Multiplication	p1 * p2	p1.__mul__(p2)
Power	p1 ** p2	p1.__pow__(p2)
Division	p1 / p2	p1.__truediv__(p2)
Floor Division	p1 // p2	p1.__floordiv__(p2)
Remainder (modulo)	p1 % p2	p1.__mod__(p2)
Bitwise Left Shift	p1 << p2	p1.__lshift__(p2)
Bitwise Right Shift	p1 >> p2	p1.__rshift__(p2)
Bitwise AND	p1 & p2	p1.__and__(p2)
Bitwise OR	p1 | p2	p1.__or__(p2)
Bitwise XOR	p1 ^ p2	p1.__xor__(p2)
Bitwise NOT	~p1	p1.__invert__()

Operator	Expression	Internally
Less than	p1 < p2	p1.__lt__(p2)
Less than or equal to	p1 <= p2	p1.__le__(p2)
Equal to	p1 == p2	p1.__eq__(p2)
Not equal to	p1 != p2	p1.__ne__(p2)
Greater than	p1 > p2	p1.__gt__(p2)
Greater than or equal to	p1 >= p2	p1.__ge__(p2)

Property

```
class Track:
  def __init__(self, artist, title, duration):
    self._artist = artist
    self._title = title
    self._duration = duration

  @property
  def artist(self): # artist instead of artist()
    return self._artist

  @property
  def title(self):
    return self._title

  @property
  def duration(self):
    return self._duration

  def __str__(self):
    return '%s - %s - %s' %(self._artist, self._title, self._duration)
```

Property

```
if __name__ == "__main__":
    track = Track("Ravi Shankar", "Bairagi Todi", 25)
    print(track)

    # access directly
    print ('%s - %s - %s' %
        (track.artist, track.title, track.duration))
```

Object Oriented Programming - II

Class, Subclass

```
class Person:  # base class
  def __init__(self, name, age): # constructor
    self._name = name
    self._age = age

  def introduce(self):              # method
    return 'Person - name: %s, age: %s'% \
     (self._name, self._age)

  def __str__(self):
    return 'Person - name: %s, age: %s'% \
     (self._name, self._age)
```

Class, Subclass

```
class Person:  # base class
  def __init__(self, name, age): # constructor
    self.__name = name
    self.__age = age
  # define name() and age() accessors

  def introduce(self):                # method
    return 'Person - name: %s, age: %s'% \
     (self.__name, self.__age)

class Student(Person): # derived class
  """ A student class
  """
  def __init__(self, name, age, id):
    super().__init__(self, name, age)
    self._id = id

  def introduce(self):  # overridden method
    return 'Student - name: %s,age: %s,id: %d'% \
           (self.name(), self.age(), self._id)
```

Class, Subclass

```
if __name__ == "__main__":
    p1 = Person('Tendulkar', 50)
    p2 = Person('Dravid', 50)
    p2.nickname = 'The Wall'

    print(p2.introduce(), \
      'nickname: ', p2.nickname)

    s1 = Student('Rahane', 32)
    print(s1.introduce())
```

Composition & Inheritance

```
# composition
class Stack:
  def __init__(self):
    self._stack = []
  def push(self,object):
    self._stack.append(object)
  def pop(self):
    return self._stack.pop()
  def length(self):
    return len(self._stack)
  def __repr__(self):
    return repr(self._stack)
```

Composition & Inheritance

```
# composition
class Stack:
  def __init__(self):
    self._stack = []
  def push(self,object):
    self._stack.append(object)
  def pop(self):
    return self._stack.pop()
  def length(self):
    return len(self._stack)
  def __repr__(self):
    return repr(self._stack)
```

```
# inheritance
class StackD(list):
  def push(self, obj):
    self.append(obj)
```

Composition & Inheritance

```
if __name__ == "__main__":
    # composition
    s = Stack()
    s.push("Multiverse")
    s.push(42)
    s.push([3,4,5])
    print(s)
    x = s.pop()
    y = s.pop()
    print(s)
```

Composition & Inheritance

```
if __name__ == "__main__":
    # composition
    s = Stack()
    s.push("Multiverse")
    s.push(42)
    s.push([3,4,5])
    print(s)
    x = s.pop()
    y = s.pop()
    print(s)

    # inheritance
    # We can use StackD the same way
    s = StackD() # etc.
```

`__str__, __repr__, __call__`

__repr__

```
class Rectangle:
  def __init__(self, w=10, h=10):
    self._width  = w
    self._height = h
  def __repr__(self):
    return 'Rectangle (width=%d,height=%d)' \
    % (self._width, self._height)
```

__repr__

```
class Rectangle:
  def __init__(self, w=10, h=10):
    self._width  = w
    self._height = h
  def __repr__(self):
    return 'Rectangle (width=%d,height=%d)' \
    % (self._width, self._height)

if __name__ == "__main__":
  r = Rectangle()
  print(r)
  print(str(r))

  r1 = Rectangle(2,2)
  print(r1)
```

__str__

```
class Rectangle:
  def __init__(self, w, h):
    self._width = w
    self._height = h
  def area(self):
    return self._width * self._height
```

__str__

```
class Rectangle:
  def __init__(self, w, h):
    self._width = w
    self._height = h
  def area(self):
    return self._width * self._height

if __name__ == '__main__':
  r1 = Rectangle(100,20)
  print(r1)
<__main__.Rectangle object at 0x7f93a8706fd0>
```

str

```
class Rectangle:
  def __init__(self, w, h):
    self._width = w
    self._height = h

  def area(self):
    return self._width * self._height

  def __str__(self):
    return '(Rectangle: %s, %s)' % \
    (self._width, self._height)
```

str

```
class Rectangle:
  def __init__(self, w, h):
    self._width = w
    self._height = h

  def area(self):
    return self._width * self._height

  def __str__(self):
    return '(Rectangle: %s, %s)' % \
    (self._width, self._height)

if __name__ == '__main__':
  r1 = Rectangle(100,20)
  print(r1)
(Rectangle: 100, 20)
```

__call__

```
class Foo:
   def __call__(self):
      print('called')

foo_instance = Foo()

# calls the __call__ method
foo_instance()
```

class & static methods

```
from datetime import date

class Person:
  def __init__(self, name, age):
      self.name = name
      self.age = age

  @classmethod
  def fromBirthYear(classname, name, year):
    return classname(name,date.today().year -year)

  @staticmethod
  def isAdult(age):
      return age > 18
```

class & static methods

```
>>> p0 = Person()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: __init__() takes exactly 3
arguments (1 given)

>>> p1 = Person('Ajinka', 21)
>>> p2 = Person.fromBirthYear('Ajinka',
1996)
>>> print p1.age
>>> print p2.age # print the result
>>> print Person.isAdult(22)
```

Built-in class attributes

```
>>> class Account():
...     ''' A class defines a new data type '''
...     pass
...
>>> dir(Account)
['__class__', '__delattr__', '__dict__', '__dir__',
'__doc__', '__eq__', '__format__', '__ge__',
'__getattribute__', '__gt__', '__hash__', '__init__',
'__init_subclass__', '__le__', '__lt__', '__module__',
'__ne__', '__new__', '__reduce__', '__reduce_ex__',
'__repr__', '__setattr__', '__sizeof__', '__str__',
'__subclasshook__', '__weakref__']
>>> print(type(Account))
<class 'type'>
>>> print(type(Account()))
<class '__main__.Account'>
```

Built-in class attributes

```
>>> Account.__name__
'Account'
>>> Account.__doc__
'A class defines a new data type'
>>> Account.__module__
'__main__'
>>> Account.__bases__
(<class 'object'>,)
```

Derived class attributes

```
>>> dir(Student)
['__doc__', '__init__', '__module__']
>>> Student.__bases__
(__main__.Person,)
>>> Student.__doc__
' A student class\n  '
```