F21SC Industrial Programming: Python Classes & Exceptions

Hans-Wolfgang Loidl

School of Mathematical and Computer Sciences,
Heriot-Watt University, Edinburgh

Semester 1 — 2021/22

[0]No proprietary software has been used in producing these slides

Outline

Class definition

- Class definition uses familiar syntax:

```
class ClassName (SuperClass_1, ..., SuperClass_n):
  statement_1
  ...
  statement_m
```

- Executing the class definition generates a class object, which can be referenced as `ClassName`.
- The expression `statement_i` generates class attributes (fields).
- Additionally, attributes of parent classes `SuperClass_i` are inherited,
- Class objects can be called just like functions (they are **callable**).
- Calling a class-object generates an **instance** of this object (no `new` necessary!).

Class attributes

- The following example generates a class with 2 attributes, one is a variable `classVar1` and one is a method `method1`.

Example

```
class C:
  "Purpose-free demo class."
  classVar1 = 42
  def method1 (self):
    "Just a random method."
    print ("classVar1 = %d" % C.classVar1)

X = C                  # alias the class object
x = X()                # create an instance of C
X.method1(x)           # call method (class view)
x.method1()            # call method (instance view)
```

- **NB:** `dir(C)` lists all attributes of a class.

Post-facto setting of class attributes

- A class is just a dictionary containing its attributes.
- Attributes can be added or modified after having created the instance (**post-facto**).
- **NB:** this is usually considered bad style!

Example

```
class D: pass          # empty class object

def method(self):      # just a function
  print (D.classVar)     # not-yet existing attribute
  print (D.__dict__['classVar'])    # same effect
  print (self.classVar)             # ditto

d = D()                # create an instance
D.method = method      # add new class attributes
D.classVar = 42
d.method()             # prints 42 (thrice)
```


Instance variables

- The following example defines a binary search tree:

Example

```
class BinTree:
  "Binary trees."
  def __init__(self, label, left=None, right=None):
    self.left = left
    self.label = label
    self.right = right
  def inorder(self):
    if self.left != None: self.left.inorder()
    if self.label != None: print (self.label)
    if self.right != None: self.right.inorder()
```

- `__init__` is a constructor that initialises its instance attributes.
- Within a method always use a qualified access as in `self.attr`.

Instance attributes

- Instance attributes can be set post-facto:

Example

```
x = C()
x.counter = 1
while x.counter < 10:
    x.counter = x.counter * 2
print (x.counter)
del x.counter
```

- `x.__class__` refers to the class-object of `x`.
- `x.__dict__` lists all attributes in `x`.
- `dir(x)` lists the namespace of `x`.

Method objects

- **Bound methods** know the instances they are working on.

```
>>> c = C()
>>> c.method1
<bound method C.method1 of <__main__.C instance at 0xb7
>>> c.method1()
```

- **Unbound methods** need the instance as an additional, first argument.

```
>>> C.method1
<unbound method C.method1>
>>> C.method1(c)
```

Inheritance

- Single inheritance:

Example

```
class EmptyTree(BinTree):
  def __init__ (self):
    BinTree.__init__ (self, None)

class Leaf(BinTree):
  def __init__ (self, label):
    BinTree.__init__ (self, label)

l1 = Leaf(6)
l1.printinorder()
```

- The constructor of the parent class has to be called explicitly.

Inheritance

- Sub-classes can add attributes.

Example

```
class MemberTree(BinTree):
  def member(self, x):
    return bool(self.label == x or
            (self.left and self.left.member(x)) or
            (self.right and self.right.member(x)))
```

- The constructor `__init__` is inherited.
- **Multiple inheritance** is possible in Python: Using `class C(C1,C2,...,Cn)` class attributes are first searched for in `C` itself, then recursively in `C1`,...,`Cn` doing a deep search.

Overloading

- Attributes in sub-classes can be over-loaded.
- In this example, if the tree is sorted, search is possible in logarithmic time:

Example

```
class SearchTree(MemberTree):
  """Ordered binary tree."""
  def member(self, x):
    return bool(self.label == x or
            (self.label > x and
             self.left and self.left.member(x)) or
            (self.label < x and
             self.right and self.right.member(x)))
```


Private Variables

- Attributes of the form `__ident` are local to the class (**private**).
- Internally they are renamed into the form `_ClassName__ident`.

Example

```
class Bla():
  __privateVar = 4
  def method(self):
    print (self.__privateVar)
    print (self.__class__.__dict__[
      '_Bla__privateVar'])

b = Bla()
b.method()              # prints 4 (twice)
```

Example: Bank Account

Example

```
class BankAccount:
  "Plain bank account."
  __latestAccountNo = 1000;  # NB: this init is done too late, when e
  def __init__(self, name, accountNo = 0, balance = 0):
    ...
  def Deposit(self, x):
      self.balance += x;
  def Withdraw(self, x):
      if self.balance >= x:
          self.balance -= x;
      else:
          raise InsufficientBalance, "Balance too low: %d" % self.bal
  def ShowBalance(self):
      print ("Current Balance: ", self.balance);
```


Example: Bank Account

Example

```
class ProperBankAccount(BankAccount):
  """Bank account with overdraft."""
  def __init__(self, name, accountNo = 0, balance = 0):
    ...
  def Withdraw(self, x):
      """Withdrawing money from a ProperBankAccount account."""
      if self.balance+self.overdraft >= x:
          self.balance -= x;
      else:
          raise InsufficientBalance, "Balance (incl overdraft) t
  def ShowAccount(self):
      """Display details of the BankAccount."""
      BankAccount.ShowAccount(self)
      print ("\t with an overdraft of ", self.overdraft)
  ...
```


Example: Bank Account

Example

```
class Tester:
    """Tester class."""
    def RunTrans(self,acct):
      """Run a sequence of transactions."""
      if (isinstance(acct,ProperBankAccount)): # test class membershi
        acct.overdraft = 200                   # if ProperBankAccount, 
      acct.ShowAccount();
      acct.ShowBalance();
      ...
      try:
        acct.Withdraw(y);
      except InsufficientBalance:
        print("InsufficientBalance ", acct.GetBalance(), " for withdra
      ...
```


Example: Bank Account

Example

```
# main:
if __name__ == '__main__':  # check whether this module is the m
    t = Tester();  # generate a tester instance

    # create a basic and a propoer account; NB: no 'new' needed
    mine = BankAccount("MyAccount");
    mineOvdft = ProperBankAccount("MyProperAccount");

    # put both accounts into a list; NB: polymorphic
    accts = [ mine, mineOvdft ]
    # iterate over the list
    for acct in accts:
        # run transactions on the current account
        t.RunTrans(acct)
```

Exceptions

- Exceptions can be caught using a `try...except...` expression.

Example

```
while True:
  try:
    x = int(raw_input("Please enter a number: "))
    break
  except ValueError:
    print ("Not a valid number.  Try again...")
```

- It is possible to catch several exceptions in one `except` block:

```
except (RuntimeError, TypeError, NameError):
  pass
```

Exceptions

- Several exception handling routines

Example

```
import sys

try:
    f = open('myfile.txt')
    s = f.readline()
    i = int(s.strip())
except IOError, (errno, strerror):
    print ("I/O error(%s): %s" % (errno, strerror))
except ValueError:
    print ("Could not convert data to an integer.")
except:
    print ("Unexpected error:", sys.exc_info()[0])
    raise
```


Exceptions: else

- If no exception was raised, the optional `else` block will be executed.

Example

```
for arg in sys.argv[1:]:
  try:
    f = open(arg, 'r')
  except IOError:
    print ('cannot open', arg)
  else:
    print (arg, 'has', len(f.readlines()), 'lines')
    f.close()
```

Raising Exceptions

- `raise Ex[, info]` triggers an exception.
- `raise` triggers the most recently caught exception again and passes it up the dynamic call hierarchy.

```
>>> try:
...     raise NameError, 'HiThere'
... except NameError:
...     print ('An exception flew by!')
...     raise
...
An exception flew by!
Traceback (most recent call last):
  File "<stdin>", line 2, in ?
NameError: HiThere
```

Clean-up

- The code in the `finally` block will be executed at the end of the current `try` block, no matter whether execution has finished successfully or raised an exception.

```
>>> try:
...     raise KeyboardInterrupt
... finally:
...     print ('Goodbye, world!')
...
Goodbye, world!
Traceback (most recent call last):
  File "<stdin>", line 2, in ?
KeyboardInterrupt
```

Exceptions: All Elements

- Here is an example of an `try` constructs with all features:

Example

```
def divide(x, y):
    try:
        result = x / y
    except ZeroDivisionError:
        print ("division by zero!")
    else:
        print ("result is", result)
    finally:
        print ("executing finally clause")
```

Pre-defined clean-up

- `with` triggers automatic clean-up if an exception is raised
- In the example below, the file is automatically closed.

Example

```
with open("myfile.txt") as f:
    for line in f:
        print (line)
```

- Using `with` is good style, because it guarantees that there are no unnecessary, open file handles around.

User-defined Exceptions

- The user can define a hierarchy of exceptions.
- Exceptions are classes, which inherit (indirectly) from the class `BaseException`.
- By default, the `__init__` method stores its arguments to `args`.
- To raise an exception, use `raise Class, instance` (`instance` is an instance of (a sub-class of) `Class`).
- Or use `raise instance` as a short-hand for:

  `raise instance.__class__, instance`
- Depending on context, `instance` can be interpreted as `instance.args`, e.g. `print instance`.

User-defined Excpetions

- The default usage of arguments can be modified.
- In this example: use the attribute `value` instead of `args`.

Example

```
class MyError(Exception):
    def __init__(self, value):
        self.value = value
    def __str__(self):
        return repr(self.value)

try:
    raise MyError(2*2)
except MyError, e:
    print ('My exception occurred, value:', e.value)
```

- Together with the constructor, the representation function `__str__` needs to be modified, too.

User-defined Exceptions

- The following code prints B, B, D (because `except B` also applies to the sub-class `C` of `B`.

Example

```
class B(BaseException):    pass
class C(B): pass
class D(C): pass

for c in [B, C, D]:
    try: raise c()
    except D: print ("D")
    except B: print ("B")
    except C: print ("C")
```


Iterators in detail

- `it = iter(obj)` returns an iterator for the object `obj`.
- `it.next()` returns the next element
- or raises a `StopIteration` exception.

Do-it-yourself Iterator

- To define an iterable class, you have to define an `__iter__()` method, which returns the next element whenever the `next()` method is called.

Example

```
class Reverse:
    "Iterator for looping over sequence backwards"
    def __init__(self, data):
        self.data = data
        self.index = len(data)
    def __iter__(self):
        return self
    def next(self):
        if self.index == 0: raise StopIteration
        self.index = self.index - 1
        return self.data[self.index]
```

Generators

- A method, containing a `yield` expression, is a **generator**.

```
def reverse(data):
    for index in range(len(data)-1, -1, -1):
        yield data[index]
```

- Generators can be iterated like this.

```
>>> for char in reverse('golf'): print (char)
...
f l o g
```


Generator Expressions

- Similar to list-comprehensions:

```
>>> sum(i*i for i in range(10))
285
>>> xvec = [10, 20, 30]
>>> yvec = [7, 5, 3]
>>> sum(x*y for x,y in zip(xvec, yvec))
260
>>> unique_words = set(word
                       for line in page
                       for word in line.split())
```


Exercises

- Go to the Python Online Tutor web page, www.pythontutor.com, and do the object-oriented programming exercises (OOP1, OOP2, OOP3).
- Implement the data structure of binary search trees, using classes, with operations for inserting and finding an element.