

Python Programing

Lesson 04: File Handling, Classes & Objects, Exception Handling

People matter, results count.

Lesson Objectives

- After completing this lesson, you will learn about:
 - File Handling
 - Class & Objects
 - Exception Handling



Working with Files

- Python supports both free form and fixed form files – text and binary
- `open()` returns a file object, and is most commonly used with two arguments: `open(filename, mode)`

- Modes:

Value	Description
'r'	Read mode
'w'	Write mode
'a'	Append mode
'b'	Binary mode (added to other mode)
'+'	Read/write mode (added to other mode)

- `f = open(r'C:\text\somelfile.txt')`
- For Input/Output: `read()`, `readline()`, `write()` and `writeline()`

Working with Files

- File Object attributes

Attribute	Description
file.closed	Returns true if file is closed, false otherwise.
file.mode	Returns access mode with which file was opened.
file.name	Returns name of the file.
file.softspace	Returns false if space explicitly required with print, true otherwise.

Classes & Objects

- Python is an object-oriented programming language, which means that it provides features that support object-oriented programming (OOP).

- Sample class definition

```
class Point:
    """ Point class represents and manipulates x,y coords. """
    def __init__(self):
        """ Create a new point at the origin """
        self.x = 0
        self.y = 0
p = Point()
print p.x, p.y
```

- Constructor: In Python we use `__init__` as the constructor name

```
def __init__(self):          # a = Point()
def __init__(self, x=0, y=0): # a = Point(5, 6)
```

Classes & Objects

- Methods

```
class Point:
```

```
    """ Point class represents and manipulates x,y coords. """
```

```
    def __init__(self, x=0): self.x = x
```

```
    def x_square(self): return self.x ** 2
```

```
p = Point(2)
```

```
print p.x_square()
```

- Objects are mutable.

Classes & Objects

■ Operator Overloading

```
class Point:
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y
    def __add__(self, other):
        return Point(self.x + other.x, self.y + other.y)
    def __mul__(self, other):
        if isinstance(other, Point):
            return Point(self.x * other.x, self.y * other.y)
        else:
            return Point(self.x * other, self.y * other)
    def __rmul__(self, other):
        return Point(self.x * other, self.y * other)
    def __repr__(self):
        return "{0}, {1}".format(self.x, self.y)
```

```
p1 = Point(2,3)
p2 = Point(3,4)
print p1 + p2    #prints (5, 7)
print p1 * p2    #prints (6, 12)
print p1 * 2     #prints (4, 6)
print 2 * p2     #prints (6, 8)
```

Classes & Objects: Operator Overloading

Operator	Special method	Operator	Special method
self + other	<code>__add__(self, other)</code>	<code>+self</code>	<code>__pos__(self)</code>
self - other	<code>__sub__(self, other)</code>	<code>abs(self)</code>	<code>__abs__(self)</code>
self * other	<code>__mul__(self, other)</code>	<code>~self</code>	<code>__invert__(self)</code> (bitwise)
self / other	<code>__div__(self, other)</code> or <code>__truediv__(self, other)</code> if <code>__future__.division</code> is active.	<code>self += other</code>	<code>__iadd__(self, other)</code>
self // other	<code>__floordiv__(self, other)</code>	<code>self -= other</code>	<code>__isub__(self, other)</code>
self % other	<code>__mod__(self, other)</code>	<code>self *= other</code>	<code>__imul__(self, other)</code>
<code>divmod(self, other)</code>	<code>__divmod__(self, other)</code>	<code>self /= other</code>	<code>__idiv__(self, other)</code> or <code>__itruediv__(self, other)</code> if <code>__future__.division</code> is in effect.
self ** other	<code>__pow__(self, other)</code>	<code>self //= other</code>	<code>__ifloordiv__(self, other)</code>
self & other	<code>__and__(self, other)</code>	<code>self %= other</code>	<code>__imod__(self, other)</code>
self ^ other	<code>__xor__(self, other)</code>	<code>self **= other</code>	<code>__ipow__(self, other)</code>
self other	<code>__or__(self, other)</code>	<code>self &= other</code>	<code>__iand__(self, other)</code>
self << other	<code>__lshift__(self, other)</code>	<code>self ^= other</code>	<code>__ixor__(self, other)</code>
self >> other	<code>__rshift__(self, other)</code>	<code>self = other</code>	<code>__ior__(self, other)</code>
<code>bool(self)</code>	<code>__nonzero__(self)</code> (used in boolean testing)	<code>self <=< other</code>	<code>__ilshift__(self, other)</code>
<code>-self</code>	<code>__neg__(self)</code>	<code>self >=> other</code>	<code>__irshift__(self, other)</code>

- Right-hand-side equivalents for all binary operators exist (`__radd__`, `__rsub__`, `__rmul__`, `__rdiv__`, ...). They are called when class instance is on r-h-s of operator:

-- `a + 3` calls `__add__(a, 3)` -- `3 + a` calls `__radd__(a, 3)`

Classes & Objects: Special methods for any class

Method	Description
<code>__init__(self, args)</code>	Instance initialization (on construction)
<code>__del__(self)</code>	Called on object demise (refcount becomes 0)
<code>__repr__(self)</code>	<code>repr()</code> and <code>`...`</code> conversions
<code>__str__(self)</code>	<code>str()</code> and print statement
<code>__sizeof__(self)</code>	Returns amount of memory used by object, in bytes (called by <code>sys.getsizeof()</code>).
<code>__format__(self, format_spec)</code>	<code>format()</code> and <code>str.format()</code> conversions
<code>__cmp__(self, other)</code>	Compares self to other and returns <0, 0, or >0. Implements >, <, == etc...
<code>__index__(self)</code>	Allows using any object as integer index (e.g. for slicing). Must return a single integer or long integer value.
<code>__lt__(self, other)</code>	Called for self < other comparisons. Can return anything, or can raise an exception.
<code>__le__(self, other)</code>	Called for self <= other comparisons. Can return anything, or can raise an exception.
<code>__gt__(self, other)</code>	Called for self > other comparisons. Can return anything, or can raise an exception.
<code>__ge__(self, other)</code>	Called for self >= other comparisons. Can return anything, or can raise an exception.
<code>__eq__(self, other)</code>	Called for self == other comparisons. Can return anything, or can raise an exception.
<code>__ne__(self, other)</code>	Called for self != other (and self <> other) comparisons. Can return anything, or can raise an exception.

Classes & Objects: Special methods for any class (contd...)

Method	Description
<code>__hash__(self)</code>	Compute a 32 bit hash code; <code>hash()</code> and dictionary ops. Since 2.5 can also return a long integer, in which case the hash of that value will be taken. Since 2.6 can set <code>__hash__ = None</code> to void class inherited hashability.
<code>__nonzero__(self)</code>	Returns 0 or 1 for truth value testing. when this method is not defined, <code>__len__()</code> is called if defined; otherwise all class instances are considered "true".
<code>__getattr__(self, name)</code>	Called when attribute lookup doesn't find name. See also __getattribute__ .
<code>__getattribute__(self, name)</code>	Same as __getattr__ but always called whenever the attribute name is accessed.
<code>__dir__(self)</code>	Returns the list of names of valid attributes for the object. Called by builtin function <code>dir()</code> , but ignored unless <code>__getattr__</code> or <code>__getattribute__</code> is defined.
<code>__setattr__(self, name, value)</code>	Called when setting an attribute (inside, don't use " <code>self.name = value</code> ", use instead " <code>self.__dict__[name] = value</code> ")
<code>__delattr__(self, name)</code>	Called to delete attribute <name>.
<code>__call__(self, *args, **kwargs)</code>	Called when an instance is called as function: <code>obj(arg1, arg2, ...)</code> is a shorthand for <code>obj.__call__(arg1, arg2, ...)</code> .
<code>__enter__(self)</code>	For use with context managers, i.e. when entering the block in a with-statement . The with statement binds this method's return value to the as object.
<code>__exit__(self, type, value, traceback)</code>	When exiting the block of a with-statement . If no errors occurred, type, value, traceback are None. If an error occurred, they will contain information about the class of the exception, the exception object and a traceback object, respectively. If the exception is handled properly, return True. If it returns False, the with-block re-raises the exception.

Classes & Objects

■ Inheritance / Sub-classing

- We can create a class by inheriting all features from another class.

The “hello” method defined in class A will be inherited by class B.

The output will be:

Hello, I'm A.

Hello, I'm A.

```
class A:
    def hello(self):
        print "Hello, I'm A."
class B(A):
    pass
a = A()
b = B()
a.hello()
b.hello()
```

- Python supports a limited form of multiple inheritance as well.
 - `class DerivedClassName(Base1, Base2, Base3):`
- Derived classes may **override methods** of their base classes.

Exception Handling

- Whenever a runtime error occurs, it creates an exception object. For example:

```
>>> print(55/0)
```

Traceback (most recent call last):

File "<interactive input>", line 1, in <module>

ZeroDivisionError: integer division or modulo by zero

- In python, the basic syntax of exception handling is

try:

some code to raise exception

except ExceptionClassName:

exception handler statements

- Example

try:

1/0

except ZeroDivisionError:

print "Can't divide anything by zero."

Exception Handling

- Below is a list of some of the built-in exceptions

Class Name	Description
Exception	The root class for all exceptions
AttributeError	Raised when attribute reference or assignment fails
IOError	Raised when trying to open a nonexistent file (among other things)
IndexError	Raised when using a nonexistent index on a sequence
KeyError	Raised when using a nonexistent key on a mapping
NameError	Raised when a name (variable) is not found
SyntaxError	Raised when the code is ill-formed
TypeError	Raised when a built-in operation or function is applied to an object of the wrong type
ValueError	Raised when a built-in operation or function is applied to an object with correct type, but with an inappropriate value
ZeroDivisionError	Raised when the second argument of a division or modulo operation is zero

Exception Handling

- Catch more than one exception
 - `except (ExceptionType1, ExceptionType2, ExceptionType3):`
- Handle multiple exceptions one-by-one
 - `except ExceptionType1: <code>`
 - `except ExceptionType2: <code>`
- Catch all exceptions
 - `except:`
- Capture the exception object
 - `except ExceptionType as e:`
- Use the `raise` statement to throw an exception
 - `raise ValueError("You've entered an incorrect value")`
- The `finally` clause of `try` is used to perform cleanup activities

Summary

- In this lesson, you learnt:
 - File Handling
 - Class & Objects
 - Exception Handling

