

An Introduction to Python

Parham Alvani

Amirkabir University of Technology

`parham.alvani@gmail.com`

May 15, 2015



Keyboard Input

- ▶ The `input()` function reads a line from `sys.stdin` and returns it with the trailing newline stripped.

```
name = input("Enter your input: ")  
print("Received input is : ", name)
```

- ▶ The open Function
- ▶ The file Object
- ▶ The close() Method
- ▶ The write() Method
- ▶ The read() Method
- ▶ See [here](#) for more details and functions.

The open Function

- ▶ Before you can read or write a file, you have to open it using Python's built-in `open()` function.
- ▶ This function creates a file object, which would be utilized to call other support methods associated with it.

```
file object = open(file_name [, access_mode] [, buffering])
```

```
fo = open("foo.txt", "w")
```

The write() Method

- ▶ The write() method writes any string to an open file.

```
fileObject.write(string)
```

```
# Open a file
```

```
fo = open("foo.txt", "wb")
```

```
fo.write( "Python is a great language.\nYeah its great!!\n")
```

```
# Close opened file
```

```
fo.close()
```

The read() Method

- ▶ The read() method reads a string from an open file.

```
fileObject.read([count])
```

```
# Open a file
fo = open("foo.txt", "r+")
str = fo.read(10);
print "Read String is : ", str
# Close opened file
fo.close()
```

The readline() Method

- ▶ Read one entire line from the file.
- ▶ A trailing newline character is kept in the string.
- ▶ If the size argument is present and non-negative, it is a maximum byte count (including the trailing newline) and an incomplete line may be returned.

```
fileObject.readline([size])
```

The readlines() Method

- ▶ Read until EOF using `readline()` and return a list containing the lines thus read.

```
fileObject.readlines([sizehint])
```


Functions

```
def square(x):  
    return x * x
```

```
def hello():  
    return "Hello"
```

```
def printme( statement ):  
    "This prints a passed string into this function"  
    print statement  
    return
```

Lambda

- ▶ Python supports simple anonymous functions through the lambda form.
- ▶ The executable body of the lambda must be an expression and can't be a statement, which is a restriction that limits its utility.

```
foo = lambda x: x * x
```



Classes

- ▶ The class statement creates a new class definition.
- ▶ The name of the class immediately follows the keyword `class` followed by a colon as follows



Classes

```
class Employee:
    """
    Common base class for all employees
    """
    empCount = 0

    def __init__(self, name, salary):
        self.name = name
        self.salary = salary
        Employee.empCount += 1

    def displayCount(self):
        print "Total Employee %d" % Employee.empCount

    def displayEmployee(self):
        print "Name : ", self.name, ", Salary: ", self.salary
```

```
emp1 = Employee("Zara", 2000)
```

Garbage Collection

- ▶ Python deletes unneeded objects automatically to free the memory space.



Inheritance

- Instead of starting from scratch, you can create a class by deriving it from a preexisting class by listing the parent class in parentheses after the new class name.



Inheritance

```
class SubClassName (ParentClass1[, ParentClass2, ...]):  
    """  
    Optional class documentation string  
    """  
    # class_suite
```

Base Overloading Methods

- ▶ `__init__(self [,args...])` : Constructor (with any optional arguments)
- ▶ `__del__(self)` : Destructor, deletes an object
- ▶ `__repr__(self)` : Evaluatable string representation
- ▶ `__str__(self)` : Printable string representation
- ▶ `__lt__(self, other)`:
- ▶ `__le__(self, other)`:
- ▶ `__eq__(self, other)`:
- ▶ `__ne__(self, other)`:
- ▶ `__gt__(self, other)`:
- ▶ `__ge__(self, other)`:

These are the so-called rich comparison methods,
and are called for comparison operators in preference to `__cmp__()` below.

Base Overloading Methods

- ▶ `__cmp__ (self, x)` : Called by comparison operations if rich comparison is not defined.
- ▶ `__add__(self, other)`:
- ▶ `__sub__(self, other)`:
- ▶ `__mul__(self, other)`:
- ▶ `__floordiv__(self, other)`:
- ▶ `__mod__(self, other)`:
- ▶ `__divmod__(self, other)`:
- ▶ `__pow__(self, other[, modulo])`:

Base Overloading Methods

- ▶ `__lshift__(self, other)`:
- ▶ `__rshift__(self, other)`:
- ▶ `__and__(self, other)`:
- ▶ `__xor__(self, other)`:
- ▶ `__or__(self, other)`:

These methods are called to implement the binary arithmetic operations
`+`, `-`, `*`, `//`, `%`, `divmod()`, `pow()`, `**`, `<<`, `>>`, `&`, `^`, `|`

- ▶ An object's attributes may or may not be visible outside the class definition.
- ▶ You need to name attributes with a double underscore prefix, and those attributes then are not be directly visible to outsiders.
- ▶ Python protects those members by internally changing the name to include the class name.
- ▶ You can access such attributes as `object._className__attrName`

Data Hiding

```
#!/usr/bin/python
```

```
class JustCounter:
    __secretCount = 0

    def count(self):
        self.__secretCount += 1
        print self.__secretCount

counter = JustCounter()
counter.count()
counter.count()
print counter.__secretCount
```

Data Hiding

1

2

Traceback (most recent call last):

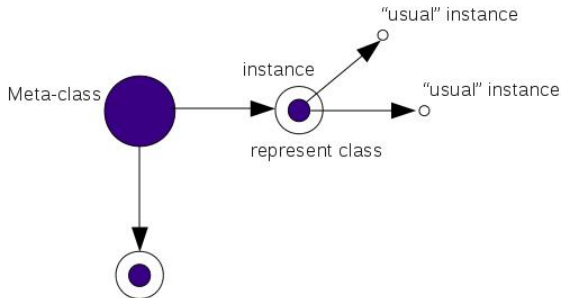
File "test.py", line 12, in <module>

print counter.__secretCount

AttributeError: JustCounter instance has no attribute '__secretCount'

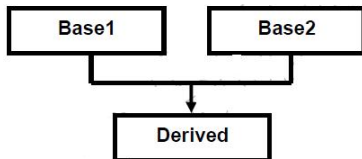
What is a metaclass in Python?

- ▶ A metaclass is the class of a class
- ▶ Like a class defines how an instance of the class behaves, a metaclass defines how a class behaves.
- ▶ A class is an instance of a metaclass.



Multiple Inheritance

- ▶ Method Resolution Order (MRO)
- ▶ C3 Algorithm



C3 linearization

- ▶ The C3 superclass linearization is an algorithm used primarily to obtain the order in which methods should be inherited (the "linearization") in the presence of multiple inheritance, and is often termed "MRO" for Method Resolution Order.

C3 Algorithm

```
class O
class A extends O
class B extends O
class C extends O
class D extends O
class E extends O
class K1 extends A, B, C
class K2 extends D, B, E
class K3 extends D, A
class Z extends K1, K2, K3
```

C3 Algorithm

$L(0) := [0]$

$L(A) := [A] + \text{merge}(L(0), [0])$
 $= [A] + \text{merge}([0], [0])$
 $= [A, 0]$

$L(B) := [B, 0]$

$L(C) := [C, 0]$

$L(D) := [D, 0]$

$L(E) := [E, 0]$

$L(K1) := [K1] + \text{merge}(L(A), L(B), L(C), [A, B, C])$
 $= [K1] + \text{merge}([A, 0], [B, 0], [C, 0], [A, B, C])$
 $= [K1, A] + \text{merge}([0], [B, 0], [C, 0], [B, C])$
 $= [K1, A, B] + \text{merge}([0], [0], [C, 0], [C])$
 $= [K1, A, B, C] + \text{merge}([0], [0], [0])$
 $= [K1, A, B, C, 0]$

C3 Algorithm

```
L(K2) := [K2] + merge(L(D), L(B), L(E), [D, B, E])
= [K2] + merge([D, 0], [B, 0], [E, 0], [D, B, E])
= [K2, D] + merge([0], [B, 0], [E, 0], [B, E])
= [K2, D, B] + merge([0], [0], [E, 0], [E])
= [K2, D, B, E] + merge([0], [0], [0])
= [K2, D, B, E, 0]
```

```
L(K3) := [K3] + merge(L(D), L(A), [D, A])
= [K3] + merge([D, 0], [A, 0], [D, A])
= [K3, D] + merge([0], [A, 0], [A])
= [K3, D, A] + merge([0], [0])
= [K3, D, A, 0]
```

C3 Algorithm

```
L(Z) := [Z] + merge(L(K1), L(K2), L(K3), [K1, K2, K3])
= [Z] + merge([K1, A, B, C, 0], [K2, D, B, E, 0], [K3, D, A, 0], [K1, K2, K3])
= [Z, K1] + merge([A, B, C, 0], [K2, D, B, E, 0], [K3, D, A, 0], [K2, K3])
= [Z, K1, K2] + merge([A, B, C, 0], [D, B, E, 0], [K3, D, A, 0], [K3])
= [Z, K1, K2, K3] + merge([A, B, C, 0], [D, B, E, 0], [D, A, 0])
= [Z, K1, K2, K3, D] + merge([A, B, C, 0], [B, E, 0], [A, 0])
= [Z, K1, K2, K3, D, A] + merge([B, C, 0], [B, E, 0], [0])
= [Z, K1, K2, K3, D, A, B] + merge([C, 0], [E, 0], [0])
= [Z, K1, K2, K3, D, A, B, C] + merge([0], [E, 0], [0])
= [Z, K1, K2, K3, D, A, B, C, E] + merge([0], [0], [0])
= [Z, K1, K2, K3, D, A, B, C, E, 0]
```

Class method differences in Python

- ▶ Bound, unbound and static
- ▶ Basically, a call to a member function a bound function is translated to a call to an unbound method.

```
class Test(object):  
    def method_one(self):  
        print "Called method_one"  
  
a_test = Test()  
  
a_test.method_one()  
# is translated to  
Test.method_one(a_test)
```

Class method differences in Python

- ▶ The `@staticmethod` tells the built-in default metaclass type (the class of a class) to not create bound methods for `method_two`.

```
class Test(object):  
    @staticmethod  
    def method_two():  
        print "Called method two"  
  
a_test = Test()  
a_test.method_two()  
# is translated to  
Test.method_two()
```

Questions?