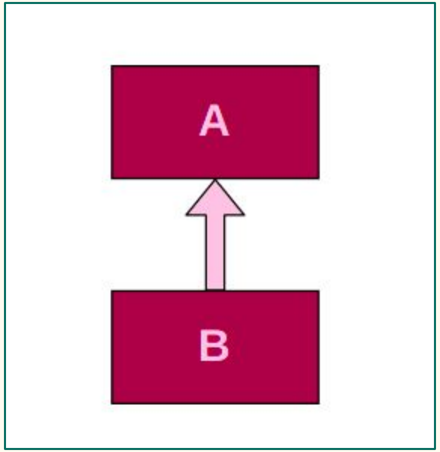
Introduction to Programming: Day 18

Soumajit Pramanik

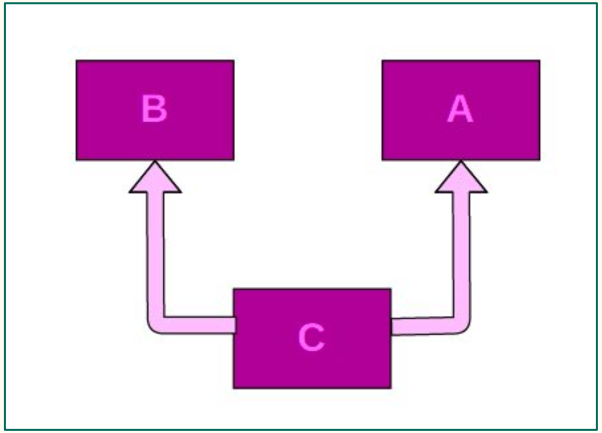
Method Resolution Order (MRO)

```
class A:
 def method(self):
   print("A.method() called")

class B(A):
 def method(self):
   print("B.method() called")

b = B()
b.method()
```



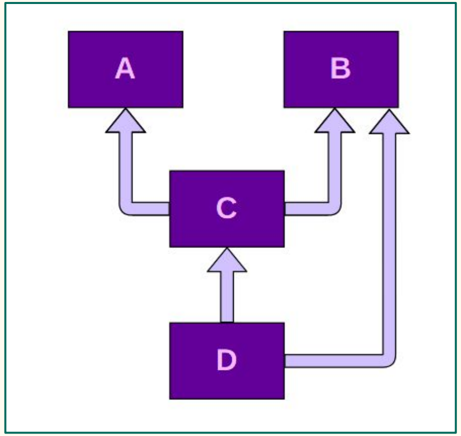
Method Resolution Order (MRO)

```
class A:
 def method(self):
   print("A.method() called")

class B:
 pass

class C(B, A):
 pass

c = C()
c.method()
```



Method Resolution Order (MRO)

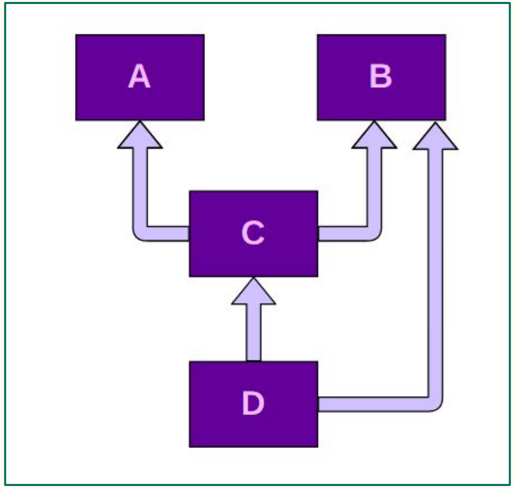
```
class A:
 def method(self):
   print("A.method() called")
class B:
 def method(self):
   print("B.method() called")
class C(A, B):
 pass
class D(B, C):
 pass

d = D()
d.method()
```



```
Traceback (most recent call last):
  File "test4.py", line 9, in <module>
    class D(B, C):
TypeError: Cannot create a consistent method resolution
order (MRO) for bases B, C
```

Method Resolution Order (MRO)

```
class A:
 def method(self):
   print("A.method() called")
class B:
 def method(self):
   print("B.method() called")
class C(A, B):
 pass
class D(C, B):
 pass

d = D()
d.method()
```



Works!!

A.method() called
D -> C -> A -> B

D.__mro__

Method Overloading

```
def product(a, b):
    p = a * b
    print(p)

def product(a, b, c):
    p = a * b*c
    print(p)
```

```
# Uncommenting the below line shows an error
# product(4, 5)

# This line will call the second product method
product(4, 5, 5)
```

Method Overloading

```
def add(datatype, *args):
    if datatype =='int':
        answer = 0
    if datatype =='str':
        answer =''
    for x in args:
        # This will do addition if the
        # arguments are int. Or
concatenation
        # if the arguments are str
        answer = answer + x
    print(answer)
```

```
# Integer
add('int', 5, 6)


# String
add('str', 'Hi ', 'All')
```

```
11
Hi All
```

Method Overloading

```
from multipledispatch import dispatch
@dispatch(int,int)
def product(first,second):
    result = first*second
    print(result)

@dispatch(int,int,int)
def product(first,second,third):
    result  = first * second * third
    print(result)

@dispatch(float,float,float)
def product(first,second,third):
    result  = first * second * third
    print(result)
```

```
product(2,3,2) #this will give
output of 12

product(2.2,3.4,2.3) # this will
give output of 17.985999999999997
```

Private variables in an instance

- many OOP approaches allow you to make a variable or function in an instance ***private***

- private means not accessible by the class user, only the class developer.

- there are advantages to controlling who can access the instance values

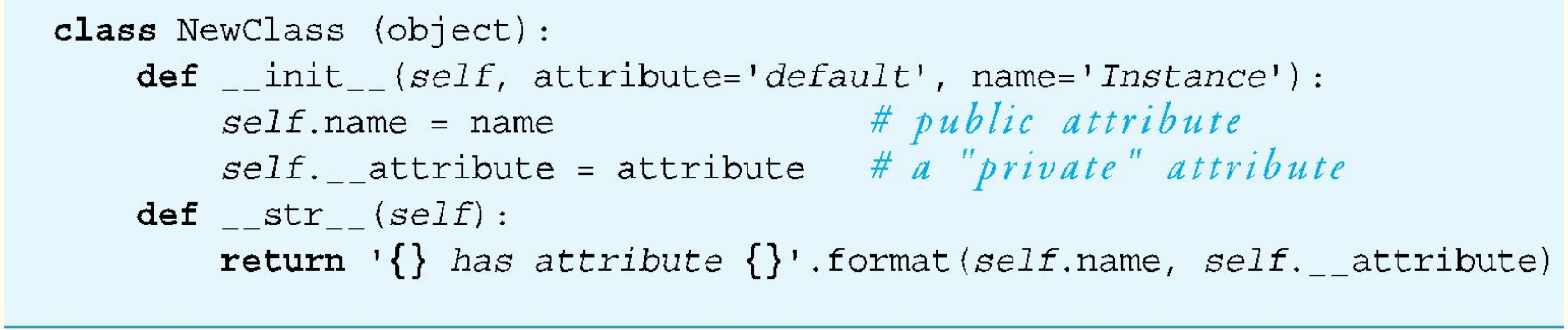
privacy in Python

- Python takes the approach “We are all adults here”. No hard restrictions.

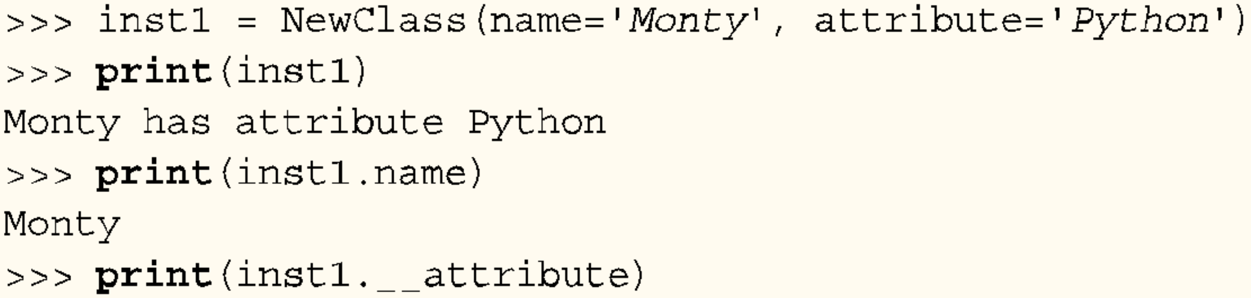
- Provides naming to avoid accidents. Use __ (double underlines) in front of any variable

- this ***mangles*** the name to include the class, namely __var becomes _class__var

privacy example



```
class NewClass (object):
    def __init__(self, attribute='default', name='Instance'):
        self.name = name                  # public attribute
        self.__attribute = attribute      # a "private" attribute
    def __str__(self):
        return '{} has attribute {}'.format(self.name, self.__attribute)
```



```
>>> inst1 = NewClass(name='Monty', attribute='Python')
>>> print(inst1)
Monty has attribute Python
>>> print(inst1.name)
Monty
>>> print(inst1.__attribute)
```

privacy example

```
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    print(inst1.__attribute)
AttributeError: 'newClass' object has no attribute '__attribute'
>>> dir(inst1)
'_NewClass__attribute', '__class__', ... , 'name']

>>> print(inst1._NewClass__attribute)
Python
```

UseCase - User Defined Exceptions

- Programs may name their own exceptions by creating a new exception class. These are derived from the Exception class, either directly or indirectly.

UseCase - User Defined Exceptions

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

    def __str__(self):
        return(repr(self.value))

try:
    raise(MyError(3*2))
 except MyError as error:
    print('A New Exception occured: ',error.value)
```

Python Modules

Python Modules

import

- When you “import” a function, for instance, you are essentially using a module

- A module is essentially a Python file with a .py extension

Using Modules

- You can import a module using import <module-name>
- and access the contents using <module-name>.<entity-name>
- You can also access entities directly

  from <module-name> import <entity-name> OR

  import <module-name>

  <alias>=<module-name>.<entity-name>

Creating Modules

Example: Create the following and save it as example.py

```
def add(x,y):
  return x+y
```

Now, in another Python file, call the add() function using the following:

```
import example
print(example.add(1,2))
```

Python Module Search Path

- PYTHONPATH is an environment variable set with the locations where the Python interpreter
- searches for modules

Typically, the module search path is defined as: PYTHONPATH=./usr/local/lib/python*X.X* which is the current directory and

/usr/local/lib/python*X.X*

Creating Modules (Contd.)

Modules have the _name_variable set to the module name

When a Python file is called as a script, the _name__is set to “__main__”. This lets you create modules that can also be executed as scripts using the following:

```
def add(x,y):
    return x+y
If__name__== “__main__”:
    print(add(1,2))
```

Packages

- Python modules can be categorized into packages by placing them within folders. The folder name becomes the package name and is used as a prefix with a period (dot) with the module name.