Object-Oriented Python II

OA Fakinlede

Contents

Purpose of Object-Oriented Programming

- Once you understand the basic object types: int, float, str, etc., and you can deliberately control the flow of execution, you are basically on your way to program in virtually any language.
- Most programming languages also afford you several pre-written code packed in different libraries that can help you achieve much with less effort.
  - You bring many of the functionality of such libraries into you code by importing them.
- You can get a lot done by simply writing code without understanding OOP paradigm.

Purpose of Object-Oriented Programming

- The Fusion 360 API, as we have seen is a large library specifically designed for 3D solid modeling, design and simulation.
  - Huge library; so big, like any API, that simply looking at the different things it has may leave you despaired as to if you can ever master using them.
- OOP is a programming paradigm
  - Rearrangement of your code, to create a structural, logical consistency,
  - Large libraries and large programs become relatively easy to use.
  - These libraries are themselves constructed in an OOP way.
  - Understanding OOP helps you to understand how they are made and how you can be effective in using them.

OOP and Large Programs

- The ability to create, extend, use, reuse and deploy large programs, libraries and Programming Interfaces require OOP logic.
  - Before we explain the Object model of the Fusion 360 API, there are some basic ideas of OOP that must be properly understood.
  - We will use simple – if not even trivial examples to make these ideas clear.
  - The ideas themselves are deep and sublime. Once you understand them, the Object model of any API will easily become transparent.
  - In today's lecture, we build on the student class we started last week, and another simple class after which we look at Fusion 360 Objects.

Types, Classes, Attributes & Methods

- Apart from the names we give to identifiers in our programs so far, perhaps the most important thing about anything we manipulate in our code, is the type it belongs to.
- You recall that what I can do with numbers 12 and 13 will depend on whether they are to be treated as integers or strings.
  - Once that is established, we can add them, may be able to multiply them, or print them to an output device or use them in some other way.
  - These are built-in types.
  - You also have lists, and other collection types. These non-scalar types are part of the core language.

Types, Classes, Attributes & Methods

- By the time you import the adsk.core, adsk.fusion, etc. and other libraries, you begin to see things like Point3D, SketchLines, SketchArcs, etc.
  - These also are types. They are types that are displayed by graphical objects on which basis you build your models.
  - What is the difference between these and ints, floats, lists, sets or dicts?
- It is VERY SIMPLE.
  - The latter are given to you in the core language,
  - The former are made in the API
  - They are all regarded, in Pythons as OBJECTS

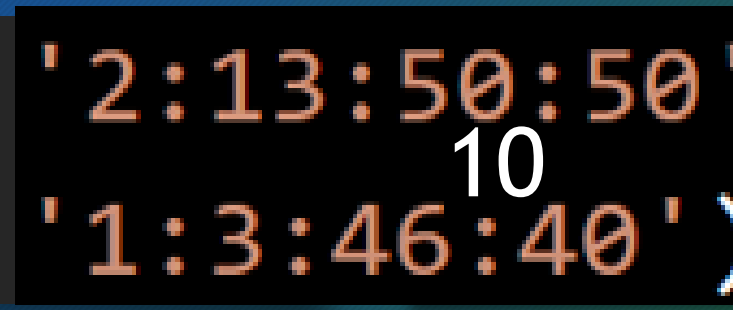
OOP Empowerment

- OOP empowers you to create, use and deploy objects.
  - Create from scratch, or based on an existing object.
  - When you do the latter, you are simply adding functionality to an existing type to make them more like what you want.
  - In OOP, you will not need to know the details of how the base objects were made for you to "inherit" its abilities and then add further abilities. You will be essentially responsible ONLY for the addition.
  - The objects you create are "First class" in the sense that the only difference in their types comes from the way they are created rather than the way they are used.
  - OOP therefore essentially empowers you to make new types, new objects

Examples: Easier than Theory

- Last week we created the student class. This we, we will add the time class to show what we mean by classes, objects, attributes and behaviour.
  - It is a good challenge to make this class more realistic. What attributes may you add: Gender, Program of study, Credits already passed, Credits remaining, Club membership, Sporting activities, Age, Height, weight, etc.
  - What methods (behaviours) may you add? Compute Good standing, IT Done? Qualified for BB team? Etc.

The New Time Class



- Simple idea. Time given in days, hours, minutes and seconds.
  - NewTime Class can turn time in minutes or secs to standard time
  - Convert standard time to minutes or secs.
  - Can add time in any form
  - Can throw exceptions when time is not correctly given
- What other things could you do? For example, add two days, 13 hours, 50 mins and 50 secs to One day, 3 hours, 46 minutes and 40 secs.
  - Can you write a program to do this?
  - We will do it by creating a NewTime Class that “knows” how to add itself!

- Initially, to create a student, we can simply make two strings and a list of six integers. The construction call in the expression,

```
Student(n3,m3,g3)
```

- Creates a student on these data values.
- Suppose we want to create a student in another way. All we have is a name. Can we do it?

```
class Student:
    def __init__(self, name, matric, grades):
        if not matric[:3].isalpha():
            raise ValueError(f'"{matric}" is not a
        if not (matric[3:].isdigit() and int(matric
            raise ValueError(f'"{matric}" is not a
        self._matric = matric
        self._name = name
        self._grades = grades
    @classmethod
    def createByName(cls, nameStr):
        ss = cls(nameStr, 'MMM000', [0]*6)
        return ss
    @property
    def matric(self):
        return self._matric
    @matric.setter
    def matric(self, value):
        self._matric = value

    @property
    def grades(self):
        return self._grades
    @grades.setter
    def grades(self, value):
        self._grades = value

    def name(self):
        return self._name

    def avg(self):
        sum = 0
        for num in self._grades:
            sum+=num
        return sum/len(self._grades)
```

```
g1 = [50,56,90,92,93,45]
g2 = [20,30,46,90,80,76]
g3 = [40,87,67,34,35,13]
g4 = [32,45,65,45,87,29]
n1 = "Asubiaro Ojukwu"
n2 = "Maroko Sango"
n3 = "Owambe Kilode"
n4 = "Miofe Mode"
m1 = "SSG123"
m2 = "MEC105"
m3 = "ELC005"
m4 = "IOT001"
stdLst = [Student(n1,m1,g1),
    Student(n2,m2,g2),
    Student(n3,m3,g3),
    Student(n4,m4,g4)]
pass
sNew = Student.createByName("Okwezilieze N
sNew.matric = 'SSG234'
mm = stdLst[0].matric
sNew.grades = [45, 56, 25, 67, 88, 99]
stdLst.append(sNew)
namesLst = {}
for std in stdLst:
    namesLst[std.avg()] = std.name()
for keys in sorted(namesLst.keys()):
    print(keys, namesLst[keys])
nuSubj = len(stdLst[0].grades)
classAvg = [0]*nuSubj
for std in stdLst:
    for i in range(nuSubj):
        classAvg[i] += std.grades[i]
for i in range(nuSubj):
    classAvg[i] /= len(stdLst)
pass
```

Examples from the API

- In Fusion 360 API, you created Point3D not simply by passing the 3 coordinates to the constructor as we have done here. You did it by passing these to a “factory” method called create. You have a statement like:

```
ptOrigin = adsk.core.Point3D.create(20, 30, 0)
```

- What is significant about this compared to simply

```
ptOrigin = adsk.core.Point3D(20, 30, 0)
```

- The factory method, instead of the simple constructor gives more flexibility. You can make several factory methods, There is only one constructor.

Instance & Class Methods

- Compare two methods:

```
31      def avg(self):
32          sum = 0
33          for num in self._grades:
34              sum+=num
35          return sum/len(self._grades)
```

```
10      @classmethod
11      def createByName(cls, nameStr):
12          ss = cls(nameStr, 'MMM000', [0]*6)
13          return ss
```

- The first is an instance method while the second is a class method.
- An instance method contains a compulsory argument, “self”. The class method, apart from the decorator @classmethod, contains a compulsory argument, “cls”.

Class Methods

```
10      @classmethod
11      def createByName(cls, nameStr):
12          ss = cls(nameStr, 'MMM000', [0]*6)
13          return ss
```


- One of the things you can do with a class method is to create a named constructor or “Factory” method. Here we used it to create a student just because we know a name only.
  - We can, later in the program supply other attributes for the same student.
  - See the flexibility here. The regular constructor, supplied by Python for us, requires us to know everything about a student before we can create one. The factory method can be more flexible. But this creates another issue:
- When we are ready, how do we supply the rest of the information?
  - We can do this by assigning to the attributes directly.
  - A more professional way to do it is the creation of getter and sette properties as we show next

Getters & Setters

- The code here creates the getter and setter properties for matric and for grades.
- The properties defined blur the difference between functions (or methods) and attributes or data.
- Is it data? Is it function? The answer is that it is one, and acts like the other. When what you have is a method, and, in this one case, you handle it as if it is simply a value.

```
@property
def matric(self):
    return self._matric
@matric.setter
def matric(self, value):
    self._matric = value

@property
def grades(self):
    return self._grades
@grades.setter
def grades(self, value):
    self._grades = value
```

Setting Values with Properties

- Statements 54 and 56 are examples of calls to the setter properties, matric and grades in the Student class.
- Using the property on the RHS of an expression is a call to the getter function.
- Making a property “read only” is achieved by refusing to write a setter function for it.
- There are other ways to achieve the same functionality, for example by writing to the internal attributes directly. The property method described here is the correct way to do it.

Enforcing Data Format

- The first four lines of the dunder init method compels a particular format for the matric.
- A ValueError exception is raised in the event of non-conformity.
- Instead of handling exceptions manually like we have done, we can place the code in a try…except block and handle all value errors together with:

```
except ValueError:
    print("Program stopped because wrong values were entered")
```

Summary for the Student Class

- The student class makes it possible to create objects of the Student Type. The class itself is the definition of the type and every Student object is created by following the rules set in the Student class
- An object of the student class can be created by a call to the the constructor with arguments passed to the initializer to create the attributes of each student created: Names, Matric, Grades, etc. These are bound to each student.
- The student class also has methods that each student instance can call to do things with each class instance. For example, you may want to find the average of a student.

Summary for the Student Class

- An instance method possesses all information needed to do things for the instance. It is limited, in this case to “self” that is, to the particular instance, the particular student, and no more.
  - In the creation of an instance method, “self” refers to the instance that calls the method. Ity therefore has the name, matric, and grades of “self” that, the student on which the method is called.
  - This means that even if an instance method appears to have no argument, it already has one: The instance calling the method!
  - std.avg() or std.grades() in line 60 is finding the avg or accessing the grades of a particular student, std.
  - An instance method may have additional arguments. Whether it does, or not, the class object, class instance that it is called for is already an argument.

Class Method

- A class method is different.
  - First, instead of “self” it has “cls” for class. It is working on behalf of the class as a whole – not just a particular instance of the class.
  - In our example here, we used a class method to create a named constructor, capable of “manufacturing” class instances.
- Remember that we have seen such factory functions already in Fusion 360 API:
  - Open any Fusion 360 API file and identify calls to factory methods.
- There other uses of class methods we shall see subsequently.

The NewTime Class

- Consider the NewTime Class shown here.
- We can create a NewTime object by a call to the constructor. We could also rely on factory methods. Identify the factory method here and imagine other factory methods that we may want to create for more flexibility
- What exception may we raise immediately? If length tt<4?

```
1  class NewTime:
2      def __init__(self, value):
3          tt = []
4          for v in value.split(':'):
5              tt.append(int(v))
6          self._secs = tt[3]
7          self._mins = tt[2]
8          self._hrs = tt[1]
9          self._days = tt[0]
10     @classmethod
11     def createBySecs(cls, value):
12         if value > 1000000:
13             raise ValueError(f'"{value}" too many seconds')
14         secs = value%60
15         res = int(value/60)
16         mins = res%60
17         res = int(res/60)
18         hrs = res%24
19         days = int(res/24)
20         val = str(days)+':'+str(hrs)+':'+str(mins)+':'+str(secs)
21         ss = cls(val)
22         return ss
23     def makeSecs(self):
24         return (((((self._days*24)+self._hrs)*60)+self._mins)*60)+self._secs
25     def __add__(self, value):
26         vv = self.makeSecs() + value.makeSecs()
27         return NewTime.createBySecs(vv)
28
29 try:
30     s1 = NewTime.createBySecs(100000)
31     s2 = NewTime('2:13:50:50')
32     s3 = NewTime('1:3:46:40')
33     s2.makeSecs()
34     s5 = s2 + s3
35     pass
36 except:
37     if ValueError:
38         print('There was a value error')
```

Constructors

```
31   ii = int('12')
32   ij = int('1s')
```

```
Exception has occurred: ValueError
invalid literal for int() with base 10: '1s'
  File "D:\Teach\Scoping001.py", line 32, in <module>
    ij = int('1s')
```


- Every type: built-in scalar, collection or class has a method having the same name as the type or class.
- This is the class constructor.
  - It is a function that takes an appropriate input, and creates an instance of the type or class. Such instances are objects of the type in question.
  - We also use the constructor as casting functions. For example, passing a string to the int constructor will attempt to create an integer out of the string, if possible.
  - The example above is a value error exception raised because the int constructor, while able to decode and make an integer out of ‘12’, was unable to do the same with ‘1s’.

Dunder Methods

- We have seen that there are instance methods as well as class methods. Constructors and factory methods are special class methods that create new instances.
  - Other important methods exist. Perhaps the most important are in the group of methods called dunder methods.
  - The portmanteau word, double underscore is used to denote the dounble underscore that pegins and ends the names of such methods.
  - The format for a dunder method for any given name is __name__. There is underscore at the beginning and at the end of the name: no spaces.
- We look at a few of them

The __init__() method

- The first thing you may want to do after creating a class instance may be to create the data that defines the state of the object. These are the data members of the object.
  - The __init__() method is designed to make this happen.
  - Once you try to create an instance, the __init__() method is executed and it usually gives life to the data attributes of the class.
  - For the class student, __init__() takes the data in the constructor call, and assigns them to _name, _matric and _grades attributes in the object. The single underscore here is to explicitly say that we do not intend to access these outside the class definition.
  - There are other housekeeping duties performed here. But the primary purpose is the initialization.

The __add__() method

- To illustrate the power of this method, consider the code here:
- Compare lines 69 and 70. We have two NewTime instances. We can create a global function to add them properly. Such a method is shown in lines 62-64.
- Inside the class definition, look at lines 58-60. It is essentially the same code. It is the method called when the regular addition operation is used between two NewTime objects:
- We have Overloaded the addition operation for this class!

```
58      def __add__(self, value):
59          vv = self.makeSecs() + value.makeSecs()
60          return NewTime.createBySecs(vv)
61
62  def addTime(t1,t2):
63      t = t1.makeSecs()+t2.makeSecs()
64      return NewTime.createBySecs(t)
65  try:
66      s1 = NewTime.createBySecs(100000)
67      s2 = NewTime('2:13:50:50')
68      s3 = NewTime('1:3:46:40')
69      s2.makeSecs()
70      s4 = addTime(s2,s3)
71      s5 = s2 + s3
```

Operator	Dunder Method	Operator	Dunder Method	Operator	Dunder Method
+	__add__(self, other)	<	__lt__(self, other)	-=	__isub__(self, other)
–	__sub__(self, other)	>	__gt__(self, other)	+=	__iadd__(self, other)
*	__mul__(self, other)	<=	__le__(self, other)	*=	__imul__(self, other)
/	__truediv__(self, other)	>=	__ge__(self, other)	/=	__idiv__(self, other)
//	__floordiv__(self, other)	==	__eq__(self, other)	//=	__ifloordiv__(self, other)
%	__mod__(self, other)	!=	__ne__(self, other)	%=	__imod__(self, other)
**	__pow__(self, other)			**=	__ipow__(self, other)
>>	__rshift__(self, other)	–	__neg__(self)	>>=	__irshift__(self, other)
<<	__lshift__(self, other)	+	__pos__(self)	<<=	__ilshift__(self, other)
&	__and__(self, other)	~	__invert__(self)	&=	__iand__(self, other)
|	__or__(self, other)			|=	__ior__(self, other)
^	__xor__(self, other)			^=	__ixor__(self, other)

Operator Overloading

- To overload an operator is to give it meaning in a different context.
  - The semantics of the overloading is your responsibility.
  - The dunder methods listed above are simply the mechanism of the overloading.
  - For example, the addition operation means different things when they occur between two ints, floats, strs or NewTime objects.
  - If the class does not define the proper dunder method, that operator is NOT defined in that context, and an exception will be raised.

Operator Overloading

- The code here will successfully execute line 71 while control will be passed to the exception handler in line 76.
- This is because, the __add__() function is defined in the class instance whereas, the __sub__() method is not defined.
- We now come to a programming principle:
  - An operation that has not been defined for the class instance cannot be performed on the objects of the class.

```
65  try:
66      s1 = NewTime.createBySecs(100000)
67      s2 = NewTime('2:13:50:50')
68      s3 = NewTime('1:3:46:40')
69      s2.makeSecs()
70      s4 = addTime(s2,s3)
71      s5 = s2 + s3
72      s6 = s2 - s3
73      pass
74  except:
75      if ValueError:
76          print('There was a value error')
77      else:
78          print('We have an unknown exception')
```

Function Overloading

- Python does not support function name overloading in the way C++ or C# does.
  - The name mangling with function arguments you have in those languages is not a viable way to do things in Python.
  - Python provides a powerful equivalent of function overloading for constructors in the way you can create factory methods for specific purposes as we have shown.
  - If you try to do further overloading by creating new functions with the same name, Python interpreter will not flag an error or raise an exception, instead, the last definition will overwrite the previous.

Summary

- By way of example, the first principle of Object-Oriented programming has been enunciated:
  - We create new types by making classes and instantiating them to new objects that have internal states and methods that define how they function and what operations are allowed on them.
  - Properties allow us to create setters and getters that access the data attributes in an orderly fashion.
  - Dunder methods help us initialize and overload operators.
  - Class methods are can be used to create factory methods and expand the ways new instances are created.

Scopes & Namespaces

- In a Python program, several dicts exist that keep track of the names identified.
- Each of these dicts constitutes a namespace.
  - You do not deal with these dicts directly but it is a good thing to understand how they work
  - They are created and kept automatically as a result of the way your program is organized.
  - The names created by all identifiers – in-built as well as those created by you, are organized into these dicts.

Names & Attributes

- Once a name is augmented by another through a dot, we can see the name after the dot as an attribute of the name before the dot.
- If you consider an augmented name as a name, then any name further augmented by it with a dot, is an attribute of the augmented name. For example, in

```
ptOrigin = adsk.core.Point3D.create(20, 30, 0)
```

The create method is an attribute of adsk.core.Point3D. Point3D is an attribute of adsk.core; core is an attribute of adsk

Uniqueness of names in a namespace

- The names in a namespace are unique and are completely unrelated to the names in another namespace.
- It is therefore important to understand where a particular name, and consequently the object it represents “lives” so that you may properly create the correct reference that can reach it.
- If you do not do this, Python can easily create a new name in the inferred namespace and you may be referring to a different object entirely

Important namespaces

- The built-in names in Python constitute a namespace. This includes all identifier keywords and functions.
- The global names in a module;
- The local names in a function invocation.
- When you create an object of, say, the student class, all the attributes and methods of that class are in the namespace of the class. So that the local names in a class constitutes a namespace.
- These attributes are reached by the name augmentation that we have noted.

Scopes

- A textual region in which the names in a namespace are directly accessible, constitutes its scope.
- A class definition, a function or method definition, are scopes.
- Clearly, scopes can contain other scopes.
  - When you are not in the scope of a named object, you cannot usually access the name directly, augmentation will be needed to reach it.
  - There are keywords that provide direct access to names that are not necessarily in scope. These are nonlocal and global keywords

Searching nested scopes

- The code in the next slide helps explain these matters concretely
- Note the effect of deleting the association and see how a hidden association replaces it.
- Best to demonstrate this by going through a debug session

```
def scopeTest():
    def doLocal():
        theName = "local name"
        return theName

    def doNonlocal():
        nonlocal theName
        theName = "nonlocal name"

    def doGlobal():
        global theName
        theName = "global name"

    theName = "test name"
    theName = doLocal()
    print("After local assignment:", theName)
    doNonlocal()
    print("After nonlocal assignment:", theName)
    doGlobal()
    print("After global assignment:", theName)
    theName = "After Everything"
class MyClass:
    """A simple example class"""
    intAttr = 23481

    def f(self):
        self._strName = "Hello, I am here"
        return self._strName
```

```
scopeTest()
iota = int('12')
print("In global scope:", theName)
xInst = MyClass()
xInst.intAttr = 1
for jota in [3,1,5,7,9]:
    xInst.intAttr *= jota
print(xInst.intAttr)
del xInst.intAttr
pass
```