AAE 875 – Fundamentals of Object Oriented Programming and Data Analytics

Cornelia Ilin, PhD

Department of Ag & Applied Economics

UW-Madison

Week 3 - Summer 2019

OOP in Python

- **OOP** is a very well known concept used to write powerful applications
- As a **data analyst** you will be required to write code to process the data
- Development in OOP is faster and cheaper; leads to **high quality** code
- With OOP you describe **how a program should operate**
- With declarative programming languages you describe what you want to accomplish without specifying how

OOP in Python

- **OOP** uses the concept of objects and classes

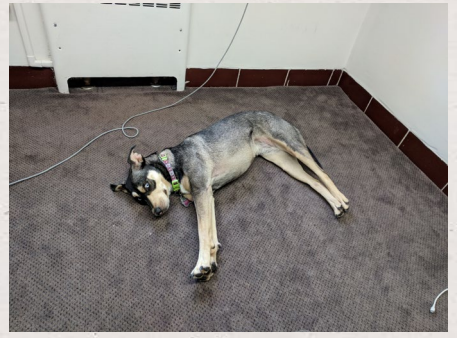
- A class can be thought as a blueprint for objects that have their own attributes (characteristics they possess), and methods (actions they perform)

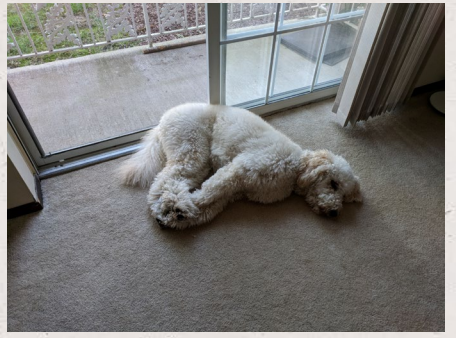
OOP in Python

- An example of a class is the class **Dog** (don't think of a particular dog)
- With a class we are trying to explain what a dog *is* and can *do*, in general
- Dogs usually have a **name** and **age.** There are called instance attributes
- Dogs can also **bark.** This is a method

OOP in Python

- Let’s talk about two dogs: Maika and Bonnie





- A specific dog is considered an object in OOP

- An object is an instance of the class **Dog**

- **This is the basic principle of OOP**

Remember Exam 1 is on Friday between 10-12 pm.
Don’t relax!

OOP in Python

- So Maika and Bonnie belong to the class **Dog**

- Their attributes are:
  - **name**: [“Maika”, “Bonnie”]
  - **age**: [2, 1]

OOP in Python

- Python is a great programming language that supports OOP
- You can use it to define attributes and methods, which you can later call
- Unlike other OOP languages (e.g. Java), it is based on dynamic typing
- So you don’t need to declare the type of variables and arguments
- Python code is easier to read and intuitive

Chapter 8: Classes

- Definition
- Constructor
- Instantiation
- Class, instance, method object
- User-defined methods
- Class vs. instance attributes

Class definition

- Classes provide a high-level approach to organize a program
- **Classes are objects** containing groups of related variables and functions
- Let's learn from an example:
  - Assume we have a database with patient info, such as: age, weight, asthma
  - We can create a class (object) *Patients* with **attributes** *age, weight, asthma*
  - This can be done using the **class** keyword and the **__init__(self)** constructor

Class definition

- Classes provide a high-level approach to organize a program
- **Classes are objects** containing groups of related variables and functions
- Let's learn from an example:
  - Assume we have a database with patient info, such as: age, weight, asthma
  - We can create a class (object) *Patients* with **attributes** *age, weight, asthma*
  - This can be done using the **class** keyword and the **__init__(self)** constructor

```
class Patients:
    def __init__(self):
        self.age = 0
        self.weight = 0  #in Kg
        self.asthma = 0  #1 if yes
```

Note: use initial capitalization for class names (e.g. Patients, PatientInfo etc)

In this example attributes are set to 0

Class constructor

- Functions defined within a class are called **methods**
- The __init__() method is a **constructor**
- The *constructor* is a *special method* with no return type and one required parameter (*self)*
- It's called when creating an **instance** of the class (instance = add new entry, e.g. new patient information)

Class constructor

- One can add additional parameters to the __init__ method (the constructor)

```
class Patients:
    def __init__(self, year, month):
        self.age = 0
        self.weight = 0  #in Kg
        self.asthma = 0  #1 if yes
```

Class constructor

- One can add additional parameters to the __init__ method (the constructor)
- These additional parameters can be added as instance attributes (can be accessed later)

```
class Patients:
    def __init__(self, year, month):
        self.age = 0
        self.weight = 0  #in Kg
        self.asthma = 0  #1 if yes
        self.year = year
        self.month = month
```

Class constructor

- One can add additional parameters to the __init__ method (the constructor)
- These additional parameters can be added as instance attributes (can be accessed later)
- Additional parameters can be set to default values

```
class Patients:
    def __init__(self, year = 2019, month = 'January'):
        self.age = 0
        self.weight = 0  #in Kg
        self.asthma = 0  #1 if yes
        self.year = year
        self.month = month
```

Class instantiation

- To define a new Patients class **variable** (i.e. add patient with corresponding health information, aka instance) use **instantiation**
- Instantiation is performed by calling the class name, similar to calling a function
- When only the required *self* parameter is present, then the class call doesn't include any arguments

```
class Patients:
    def __init__(self):
        self.age = 0
        self.weight = 0  #in Kg
        self.asthma = 0 #1 if yes

patient1 = Patients()
```

class instantiation

Class instantiation

- To define a new Patients class **variable** (i.e. add patient with corresponding health information, aka instance) use **instantiation**
- Instantiation is performed by calling the class name, similar to calling a function
- When additional parameters are present (with no default values), then the class call includes arguments for the additional parameters

```
class Patients:
    def __init__(self, year, month):
        self.age = 0
        self.weight = 0  #in Kg
        self.asthma = 0  #1 if yes
        self.year = year
        self.month = month

patient1 = Patients(2017, 'January')
```

class instantiation when additional parameters with no default values are present

Class instantiation

- To define a new Patients class **variable** (i.e. add patient with corresponding health information, aka instance) use **instantiation**
- Instantiation is performed by calling the class name, similar to calling a function
- When additional parameters are present (with default values), then the class call doesn't include arguments for the additional parameters

```
class Patients:
    def __init__(self, year = 2019, month = 'January'):
        self.age = 0
        self.weight = 0  #in Kg
        self.asthma = 0  #1 if yes
        self.year = year
        self.month = month

patient1 = Patients()
```

class instantiation when additional parameters with default values are present

Class instantiation

- To define a new Patients class **variable** (i.e. add patient with corresponding health information, aka instance) use **instantiation**
- Instantiation is performed by calling the class name, similar to calling a function
- One can add a mix of additional parameters w/ and w/o default values. Arguments w/o default values must come first, and must be in order!

```
class Patients:
    def __init__(self, month, year = 2019):
        self.age = 0
        self.weight = 0  #in Kg
        self.asthma = 0  #1 if yes
        self.year = year
        self.month = month

patient1 = Patients('January')
```

Class instantiation

- To define a new Patients class **variable** (i.e. add patient with corresponding health information, aka instance) use **instantiation**
- Instantiation is performed by calling the class name, similar to calling a function
- The instantiation operation automatically calls the constructor (__init__ method)

```
class Patients:
    def __init__(self):
        self.age = 0
        self.weight = 0  #in Kg
        self.asthma = 0 #1 if yes

patient1 = Patients()
```

instantiation calls the constructor to create a new instance (self) of the class

Class instantiation

- To define a new Patients class **variable** (i.e. add patient with corresponding health information, aka instance) use **instantiation**
- Instantiation is performed by calling the class name, similar to calling a function
- The instantiation operation automatically calls the constructor (__init__ method)
- The required parameter of the __init__ method (*self*) references each new instance created

```
class Patients:
    def __init__(self):
        self.age = 0
        self.weight = 0  #in Kg
        self.asthma = 0 #1 if yes

patient1 = Patients()
```

Class instantiation

- To define a new Patients class **variable** (i.e. add patient with corresponding health information, aka instance) use **instantiation**
- Instantiation is performed by calling the class name, similar to calling a function
- The instantiation operation automatically calls the constructor (__init__ method)
- The required parameter of the __init__ method (*self*) references each new instance created

```
class Patients:
    def __init__(self):
        self.age = 0
        self.weight = 0  #in Kg
        self.asthma = 0 #1 if yes

patient1 = Patients()

# set attribute values for the instance created above
patient1.age, patient1.weight, patient1.asthma = 30, 60, 0
```


attributes are accessed using the **dot notation**

Top Hat Question # 1

What is a class?

Answer:

Top Hat Question # 2

What is __init__?

Answer:

Top Hat Question # 3

What is an instance of a class?

Answer:

Top Hat Question # 4

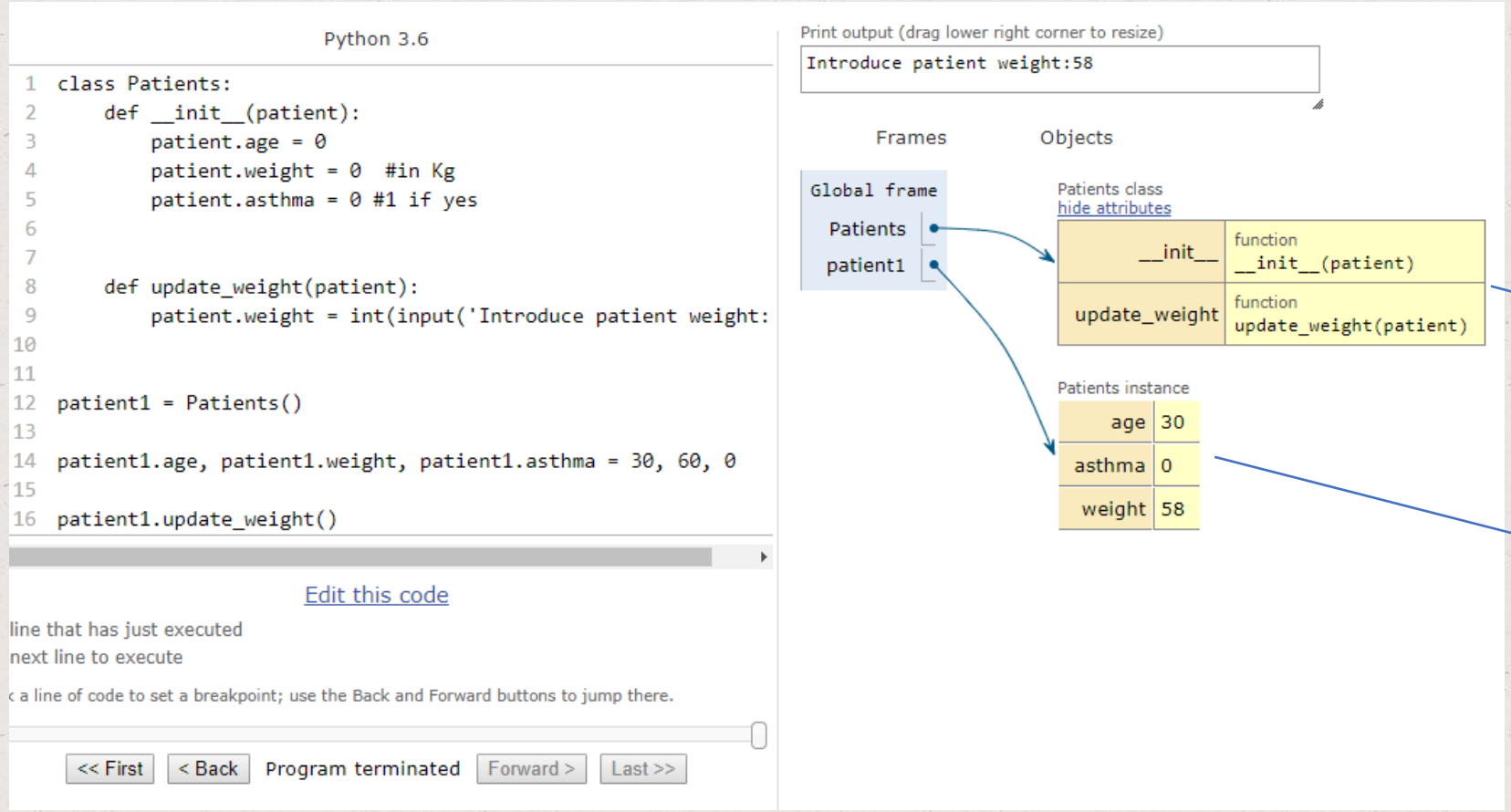
What is the output?

```
class Patients:
    def __init__(self, year = 2020, month):
        self.age = 0
        self.weight = 0  #in Kg
        self.asthma = 0  #1 if yes
        self.year = year
        self.month = month

patient1 = Patients('January')
print(patient1.month)
```

Class vs. instance object

- A **class object** creates new class instances
- An **instance object** represent a single instance of a class



User-defined methods

- A class definition may include user-defined methods
- Example: update patient weight after each visit

```
class Patients:
    def __init__(self):
        self.age = 0
        self.weight = 0  #in Kg
        self.asthma = 0 #1 if yes

    def update_health(self):
        self.weight = int(input('Introduce patient weight: '))
```

User-defined methods

- A class definition may include user-defined methods
- Example: update patient weight after each visit

```
class Patients:
    def __init__(self):
        self.age = 0
        self.weight = 0  #in Kg
        self.asthma = 0 #1 if yes

    def update_health(self):
        self.weight = int(input('Introduce patient weight: '))
```

- The parameter of the user-defined method has to match the required parameter of the constructor method!

User-defined methods

- How do we call a user-defined method? (E.g. how do we update the weight of a patient?)

```
class List:
    def __init__(self):
        self.age = 0
        self.weight = 0  #in Kg
        self.asthma = 0 #1 if yes

    def update_health(self):
        self.weight = int(input('Introduce patient weight: '))

patient1 = Patients()
patient1.age = 30
patient1.weight = 60
patient1.asthma = 0

patient1.update_health()
```


No argument was provided!

Top Hat Question # 5

What is patient1's asthma status if user input is 1?

```
class Patients:
    def __init__(self):
        self.age = 0
        self.weight = 0  #in Kg
        self.asthma = 0 #1 if yes

    def update_health():
        self.asthma = int(input('Introduce patient asthma status (1 = yes): ')

patient1 = Patients()
patient1.age = 30
patient1.weight = 60
patient1.asthma = 0
patient1.update_health()
```

Top Hat Question # 6

Is the method update_health() correctly defined?

```
class Patients:
    def __init__(self, year):
        self.age = 0
        self.weight = 0  #in Kg
        self.asthma = 0 #1 if yes
        self.year = year

    def update_health(self):
        self.asthma = int(input('Introduce patient asthma status (1 = yes): ')

patient1 = Patients()
patient1.age = 30
patient1.weight = 60
patient1.asthma = 0
patient1.update_health()
```

Class vs. Instance attributes

- A **class attribute** is shared among all instances of that class
  - Defined within the scope of the class
- An **instance attribute** can be unique to each instance
  - Defined using dot notation from within a method or from outside of the class scope
  - When using dot notation the instance namespace is searched first followed by the class namespace

```
class Patients:
    year = 2018
    def __init__(self):
        self.age = 0
        self.weight = 0  #in Kg
        self.asthma = 0 #1 if yes

patient1 = Patients()
patient2 = Patients()
patient1.age, patient1.weight, patient1.asthma = 30, 60, 0
patient2.age, patient2.weight, patient2.asthma = 28, 55, 1
```

Class attribute

Instance attributes

Class vs. Instance attributes

- A **class attribute** is shared among all instances of that class
  - Defined within the scope of the class
- An **instance attribute** can be unique to each instance
  - Defined using dot notation from within a method or from outside of the class scope
  - When using dot notation the instance namespace is searched first followed by the class namespace

- Good practice: avoid using same names for class and instance attributes!!

Class vs. Instance attributes

- How we can use class attributes?

```
class Patients:
    year = 2018
    def __init__(self):
        self.age = 0
        self.weight = 0  #in Kg
        self.asthma = 0 #1 if yes

patient1 = Patients()
patient2 = Patients()
patient1.age, patient1.weight, patient1.asthma = 30, 60, 0
patient2.age, patient2.weight, patient2.asthma = 28, 55, 1
print(patient1.year)
patient1.year = 2019
print(patient1.year)
```

Top Hat Question # 7

How many attributes does patient1 has? How many patient2?

```
class Patients:
    year = 2018
    def __init__(self):
        self.age = 0
        self.weight = 0  #in Kg
        self.asthma = 0 #1 if yes

patient1 = Patients()
patient2 = Patients()
patient1.age, patient1.weight, patient1.asthma = 30, 60, 0
patient2.age, patient2.weight, patient2.asthma = 28, 55, 1
print(patient1.year)
patient1.year = 2019
print(patient1.year)
```

An example w/ classes

```
class Student(object):
    def __init__(self, name, age, gender, level, grades=None):
        self.name = name
        self.age = age
        self.gender = gender
        self.level = level
        self.grades = grades or {}

    def setGrade(self, course, grade):
        self.grades[course] = grade

    def getGrade(self, course):
        return self.grades[course]

    def getGPA(self):
        return sum(self.grades.values())/len(self.grades)

# Define some students
john = Student("John", 12, "male", 6, {"math":3.3})
jane = Student("Jane", 12, "female", 6, {"math":3.5})

# Now we can get to the grades easily
print(john.getGPA())
print(jane.getGPA())
```

Same example w/ dictionaries

```
def calculateGPA(gradeDict):
    return sum(gradeDict.values())/len(gradeDict)

students = {}
# We can set the keys to variables so we might minimize typos
name, age, gender, level, grades = "name", "age", "gender", "level", "grades"
john, jane = "john", "jane"
math = "math"
students[john] = {}
students[john][age] = 12
students[john][gender] = "male"
students[john][level] = 6
students[john][grades] = {math:3.3}

students[jane] = {}
students[jane][age] = 12
students[jane][gender] = "female"
students[jane][level] = 6
students[jane][grades] = {math:3.5}

# At this point, we need to remember who the students are and where the grades are stored. Not a huge deal, but avoided by OOP.
print(calculateGPA(students[john][grades]))
print(calculateGPA(students[jane][grades]))
```

References

- OOP in Python:

https://www.datacamp.com/community/tutorials/python-oop-tutorial

- An example w/ and w/o classes:

https://stackoverflow.com/questions/33072570/when-should-i-be-using-classes-in-python