DS-GA 1007 | Lecture 2

Programming for Data Science

Jeremy Curuksu, PhD
NYU Center for Data Science
jeremy.cur@nyu.edu

February 10, 2023

DS-GA 1007 Curriculum

Programming for Data Science:

- Introduction to Programming in Python
- **Best Practice Programming and Software Engineering**
- Program Efficiency
- Interacting with Programs
- Array Manipulation for Scientific Computing
- Data Visualization
- Advanced Data Objects ($\times 4$)
- Environments for Collaborative Programming
- Industrial Applications

Best Practice Programming in Python

Best Practice Programming in Python

Last week:

- What is programming?
- Primitive Data Types and Control Flow
- Compound Data Types: Tuples, Lists, Dictionaries

Today:

- **Modularization and Abstraction**
- **Functions and Objects in Python**
- **Testing and Debugging Programs**

Modularization and Abstraction

Programming Best Practices

- **Measured by amount of functionality**, not volume
- **Modularization:** Decompose code in self-contained modules to keep code organized and coherent
- **Abstraction:** **Define modules as functions or objects which can be reused**. Leverage preexisting modules, either built-in or imported (NumPy, Pandas, ...)
- **Document** each module (input, output, properties)
- **Test and debug modules individually**, pre-emptively

Functions in Python

Decomposition with Functions

- **Decompose to create structure:** Functions should be self-contained and reusable
- **Abstract to suppress details:** Function should come with specifications or docstrings. A function can then be used as a black box: no need to see details
- **Functions are not run until they are invoked**
- **Functions have:**
  - **A name**
  - **A body**
  - A docstring (optional)
  - Input parameters (optional)
  - Returns outputs (optional)

Decomposition with Functions

Function

```
def <function name>(<parameters>):
    """ DocString """
    <body>
    return <output>
```

Example:

```
def testeven(i):
    """ Input: A positive integer number
        Output: True if number is even, False if odd
    """
    print("Executing test even for i = {}".format(i))
    return i%2 == 0
```

Scope of Variables in Functions

- **Scope:** A new ”*local*” scope for variable names is created when the program enters a function
- **Local Scope:**
  - Variables defined inside a function are not defined outside the function
  - Values fetched to function parameters get bound to local variable names when the function is invoked
- **Global Scope:**
  - Variables defined outside *can be accessed* inside (”*visible everywhere*”)
  - Variables defined outside *cannot be modified* inside

Local Variables in Functions

```
def f(x):
    """Return square of a number"""
    x = x**2
    return x

y = 10
f(y)
print(y)
```

Output: ?

Local Variables in Functions

```
def f(x):
    """Return square of a number"""
    x = x**2
    return x

y = 10
f(y)
print(y)
```

Output: 10

Local Variables in Functions

```
def f(x):
    """Return square of a number"""
    x = x**2
    return x

x = 10
f(x)
print(x)
```

Output: ?

Local Variables in Functions

```python
def f(x):
    """Return square of a number"""
    x = x**2
    return x

x = 10
f(x)
print(x)
```

Output: 10

Local Variables in Functions

```
def f(x):
    """Return square of a number"""
    x = x**2
    return x

x = 10
x = f(10)
print(x)
```

Output: 100

Local Variables in Functions

```
def f(x):
    """Return square of a number"""
    x = x**2
    return x

x = 10
x = f(x)
print(x)
```

Output: 100

Global Variables in Functions

```
def f(x):
    x = x**2
    x = x + y
    return x

y = 10
x = f(10)
print('x = {}, y = {}'.format(x,y))
```

Output: $x = 110, y = 10$

Global Variables in Functions

```python
def f(x):
    """Return square of a number"""
    y = x**2
    return y

y = 10
x = f(10)
print('x = {}, y = {}'.format(x,y))
```

Output: $x = 100, y = 10$

Global Variables in Functions

```
def f(x):
    x = x**2 + y
    y = x
    return y

y = 10
x = f(10)
print('x = {}, y = {}'.format(x,y))
```

Output: *UnboundLocalError*

Input and Ouput to Functions

Input: parameters

- Optional, from 0 to 256
- Can be any data type fit to operations happening inside the function
- Can be other functions
- Can be value or existing global variable name
- Global variables input as parameters get reassigned to local variable names

Output: return or print

- Optional
- Can use only 1 return with only 1 value per function, but can be any data type
- Can use as many print statements as desired
- If no return given, Python returns the value *None*
- Global variables are not impacted inside functions

Objects in Python

Decomposition with Objects

What are Objects?

- **Objects are data abstraction**. An object is a collection of:
  - **Data Attributes**: Internal representation of the object (equivalent to *data variables*)
  - **Methods**: Interface for interacting with the object (equivalent to *functions*)
- **Objects are instances of a class**
  - Class statements are blueprint to create objects
  - Once an object is created, its attributes and methods are accessed by the dot operator: *objectname.attributename*()

Decomposition with Objects

Object

```
class name(<superclass>):
    """ DocString """
    <body>
```

Example:

Define new type of object:

```
class dog(animal):
    speed = 30
    race = 'Not specified'
    domesticated = True
```

Use the object:

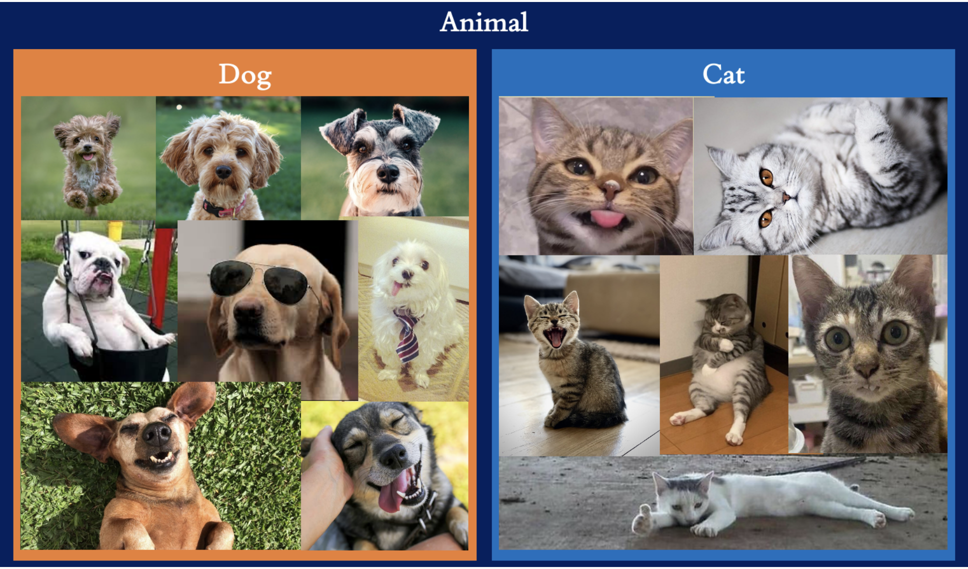
```
fido = dog() # Create instance
print(fido.domesticated)
fido.speed = 20
fido.race = 'Schnauzer'
fido.cuteness = 'XXL'
```

Object Oriented Programming (OOP)

Advantages of OOP for Data Science

- **Create new types (=classes) of data objects**
  - Create your own data *types* with custom attributes
  - Bundle together objects of common attributes
- **Modularization** $=>$ **Divid & Conquer code development**
  - Implement and test behavior of each class separately
  - Access attributes and methods consistently using the dot operator: No collision on variable and function names
- **Abstraction** $=>$ **Easy to reuse code**
  - Separate implementation of *defining* vs. *using* an object
  - Inheritance: Build layers of object abstractions that inherit behaviors from other classes of objects

Object Oriented Programming (OOP)



Attributes and Methods of Objects

Attributes and Methods are accessed by the "." operator

- **Data Attributes**
  - Data objects that make up the class
  - "What it is"

  Example: Class *2Dcoordinate* made up of two numbers

- **Methods** *(Procedural Attributes)*
  - Functions that only work with this class
  - "What it does"

  Example: Class *2Dcoordinate* with a method to compute the distance between two *2Dcoordinate* objects

Methods of Object

Defining and Invoking Methods

- When creating a method, the first argument passed to it (called *self*) must be the object itself
- Invoke the method with "." and Python automatically fills in *self*
- Other than *self* and ".", methods behave exactly like functions

Define new type of object:

```
class coordinate(object):
   x = 0.1
   y = 0.1
   def distance(self,other):
      dx2 = (x-other.x)**2
      dy2 = (y-other.y)**2
      return (dx2 + dy2)**0.5
```

Use the object:

```
p1 = coordinate()
p2 = coordinate()
p1.x = 0.8
p1.y = 0.2
d = p1.distance(p2)
print('Distance: ',d)
```

The Object Initialization Method

The __ *init*__ Method

- A special method called __ *init*__ can be created to **initialize data attributes when creating an instance of the class**
- Python automatically calls __*init*__ when creating an object
- Out of scope for this course: Implement *set* and *get* methods to access *all* data attributes of an object

Define new type of object:

```
class coordinate(object):
   def __init__(self,a,b):
      self.x = a
      self.y = b
```

Use the object:

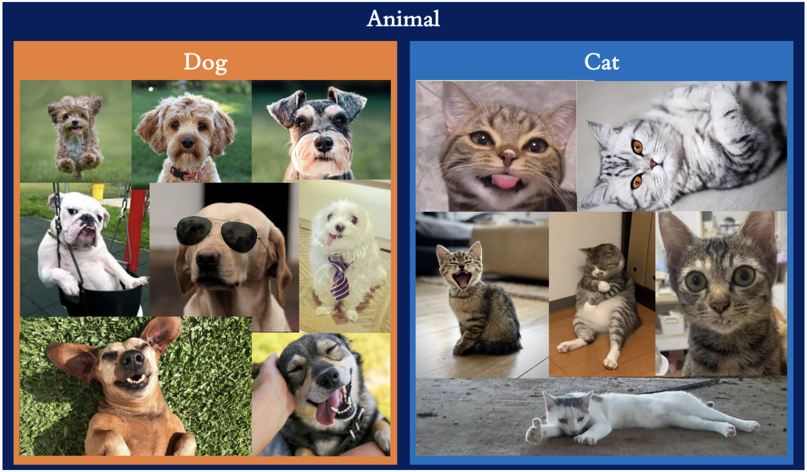
```
p1 = coordinate(0.8,0.2)
p2 = coordinate(0.5,0.5)
d = p1.distance(p2)
print('Distance: ',d)
```

Example of New Data Object

```
class coordinate(object):
   def __init__(self,a=0.1,b=0.1):
      self.x = a
      self.y = b
   def distance(self, other):
      dx2 = (self.x-other.x)**2
      dy2 = (self.y-other.y)**2
      return (dx2 + dy2)**0.5
p1 = coordinate(0.8,0.2)
p2 = coordinate(0.5,0.5)
d = p1.distance(p2)
print('Distance: ',d)
```

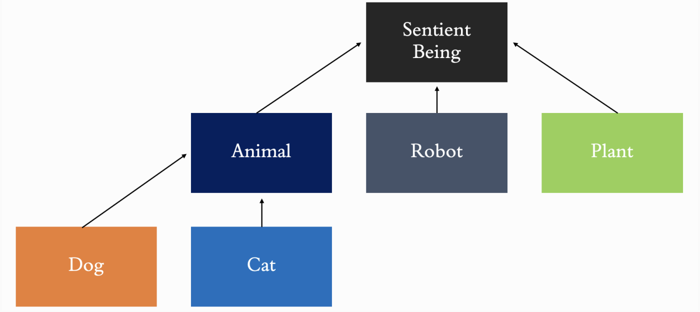
Inheritance of Object Types

- **Parent class = Superclass**
  - In Python all classes derive from a superclass



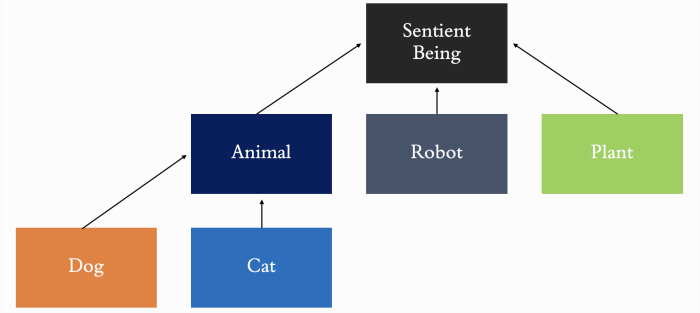
Inheritance of Object Types

- **Parent class = Superclass**
  - In Python all classes derive from the parent class *Object*
- **Child class = Subclass**
  - Inherits all data and behaviors from parent class
  - Add more information and more behavior
  - Override behavior



Inheritance of Object Types

- **For *any* data object in Python, you have access to methods in current class definition and up the hierarchy**
- **You can only use the first method up the hierarchy with that method name**: A class may have method with same name as in superclass, but it overrides inherited method with same name



Testing and Debugging Programs

Testing and Debugging Programs

- **Defensive Programming**:
  - **Modularization**: Break up programs into modules (functions, objects) and leverage pre-existing modules
  - **Abstraction**: Document specification for each module (expected input-output, assumptions on code design)
- **Test and debug modules individually**:
  - **Test**: Identify where and when errors happen
  - **Debug**: Understand and solve errors
  - **Pre-emptively...** Implement handles to **raise exceptions** and **assert** module specifications

Testing Programs

Testing $=$ Identifying where and when errors happen

- **What to test?**
  - **Unit test**: Test/validate each piece of program separately
  - **Regression test**: Add test for bugs as you find them
  - **Integration test**: Ensure overall program runs
- **How to test?**
  - **Intuitive or Random test**: Explore possible inputs and test cases randomly or based on intuition
  - **Black Box test**: Explore test cases through specification
  - **Glass Box test**: Explore test cases through code

Black Box Testing

Design test cases without looking at code

- Only use specification to define typical values and edge cases
- Can be reused if implementation changes
- Can be done by someone other than the implementer

Example: How would you test this function?

```python
def topgrade(l):
    '''Read list of grades and return highest one'''
    top = l[0]
    for r in l[1:]:
        if r > top: top = r
    return top
```

Glass Box Testing

Design test cases based on the code

- Use code to test all possible branchings, loops, etc
- A program is called *path-complete* if every potential scenario through code has been tested at least once
- Can run loops arbitrarily many times yet miss key scenarios

Example: How would you test this function?

```python
def topgrade(l):
    '''Read list of grades and return highest one'''
    top = l[0]
    for r in l[1:]:
        if r > top: top = r
    return top
```

Debugging Programs

Debugging = Understanding and solving errors

- **The scientific method**:
  - Study code and data to form hypotheses on origin of bug
  - Use repeatable experiments
- **The tools**:
  - **Built-in** Python editors and interpreters
  - **Print statements** to test hypotheses. Quickly locate bugs with prints at begin/end of functions (or use *bisection*)
  - **Systematic**: Make change, compare new *vs.* old, deduce, ...

Common Errors in Python

```
l = ['a','b','c']
l[3]
```

▶ ?

```
int(l)
l[2]/4

c/4

len(['a','b','c']

open('c.dat')
```

Common Errors in Python

```
l = ['a','b','c']
l[3]                    ▶ ?

int(l)
l[2]/4

c/4

len(['a','b','c']

open('c.dat')
```

Common Errors in Python

```
l = ['a','b','c']
l[3]
```

- **IndexError**: E.g., Trying to access beyond the limits of a list

```
int(l)
l[2]/4
```

- **?**

```
c/4
```

```
len(['a','b','c']
```

```
open('c.dat')
```

Common Errors in Python

```
l = ['a','b','c']
l[3]
```

- **IndexError**: E.g., Trying to access beyond the limits of a list

```
int(l)
l[2]/4
```

- **TypeError**: E.g., Trying to convert an inappropriate type (list $>$ int)

```
c/4
```

```
len(['a','b','c']
```

```
open('c.dat')
```

Common Errors in Python

```
l = ['a','b','c']
l[3]
```

- **IndexError**: E.g., Trying to access beyond the limits of a list

```
int(l)
l[2]/4
```

- **?**

```
c/4
```

```
len(['a','b','c']
```

```
open('c.dat')
```

Common Errors in Python

```
l = ['a','b','c']
l[3]

int(l)
l[2]/4

c/4

len(['a','b','c']

open('c.dat')
```

- **IndexError**: E.g., Trying to access beyond the limits of a list
- **TypeError**: E.g., Trying to convert an inappropriate type, mixing inappropriate data types, etc
- **?**

Common Errors in Python

```
l = ['a','b','c']
l[3]
```

- **IndexError**: E.g., Trying to access beyond the limits of a list

```
int(l)
l[2]/4
```

- **TypeError**: E.g., Trying to convert an inappropriate type, mixing inappropriate data types, etc

```
c/4
```

- **NameError**: E.g., Referencing a non-existent variable

```
len(['a','b','c']
```

- **?**

```
open('c.dat')
```

Common Errors in Python

```
l = ['a','b','c']
l[3]
```

- **IndexError**: E.g., Trying to access beyond the limits of a list

```
int(l)
l[2]/4
```

- **TypeError**: E.g., Trying to convert an inappropriate type, mixing inappropriate data types, etc

```
c/4
```

- **NameError**: E.g., Referencing a non-existent variable

```
len(['a','b','c']
```

- **SyntaxError**: E.g., Forgetting to close parenthesis, quotation, etc

```
open('c.dat')
```

- **?**

Common Errors in Python

```python
l = ['a','b','c']
l[3]
```

- **IndexError**: E.g., Trying to access beyond the limits of a list

```python
int(l)
l[2]/4
```

- **TypeError**: E.g., Trying to convert an inappropriate type, mixing inappropriate data types, etc

```python
c/4
```

- **NameError**: E.g., Referencing a non-existent variable

```python
len(['a','b','c']
```

- **SyntaxError**: E.g., Forgetting to close parenthesis, quotation, etc

```python
open('c.dat')
```

- **IOError**: File not found

Pre-emptively Handling Errors

Exception Handlers

try/except/raise: **Try** a block of code, if execution hits unexpected condition, handle this **excep**tion with specific instructions, or **raise** an error which stops execution

Example:

```
def squareroot(x):
    try:
        return(math.sqrt(x))
    except ValueError:
        print("Warning: Input is not positive")
        return(math.sqrt(-x))
    except:
        raise TypeError("Stopped: Input is not a number")
```

Pre-emptively Handling Errors

Assertion Handlers

assert: Stop execution and raise error if assumptions are not met. This pre-emptively locates sources of bugs as soon as introduced and avoid propagating them (*defensive programming*)

Example:

```
def ratio(x,y):
    assert y != 0,'Denominator of ratio is zero'
    return x/y
```

Thank you!