UNIVERSITÄT BIELEFELD
Faculty of Technology

# Programming

## Object oriented Programming

Harsha Manjunath

Faculty of Technology, Bielefeld University

Loops

Functions

**Classes, Modules & Packages**

Programming Errors & Debugging

# Creating new types

- A `class` defines a new type
- It can provide
  - class variables & functions
  - instance variables & functions

# Classes—example of code reuse

```
1  class Library:
2      description = 'This is a Library'
3
4      def __init__(self, name):
5          # name the library
6          self.name = name
7          # create empty book storage on initialization
8          self.storage = list()
9
10     def addBook(self, book):
11         self.storage.append(book)
12
13     def getAllBooks(self):
14         return tuple(self.storage)
15
16 myLib = Library('Bodleian Library')
17 myLib.addBook('The Art of Computer Programming (D. Knuth)')
```

# Modules

- Every `.py` file is a module
- Modules can host functions, variables, and classes
- Imported modules with `import` statement
- Should not have blocks of code that are immediately executed
- Explicit reference to module scope: `global`
- Name of module available as global variable `__name__`

# Modules—example of code reuse

## mystringutils.py

```
#
# A module for all kinds of string utils
#

def findSubstringInStrings(stringCollection,
      pattern):
    occ = list()
    for i, s in enumerate(stringCollection):
        j = s.find(pattern)
        while j != -1:
            occ.append((i, j))
            j = s.find(pattern, j+1)
    return occ
```

## myscript.py

```
#!/usr/bin/env python3

import mystringutils

if __name__ == '__main__':
    myStringList = ['the rain in spain',
        'ain\'t no sunshine',
        'she was greeted with disdain']

    occOfAin = mystringutils.
        findSubstringInStrings(myStringList,
         'ain')
    print(occOfAin)
```

# Modules—example of code reuse

## mystringutils.py

```
#
# A module for all kinds of string utils
#

def findSubstringInStrings(stringCollection,
      pattern):
    occ = list()
    for i, s in enumerate(stringCollection):
        j = s.find(pattern)
        while j != -1:
            occ.append((i, j))
            j = s.find(pattern, j+1)
    return occ
```

## myscript.py

```
#!/usr/bin/env python3

import mystringutils as su

if __name__ == '__main__':
    myStringList = ['the rain in spain',
        'ain\'t no sunshine',
        'she was greeted with disdain']

    occOfAin = su.findSubstringInStrings(
        myStringList, 'ain')
    print(occOfAin)
```

# Modules—example of code reuse

## mystringutils.py

```
#
# A module for all kinds of string utils
#

def findSubstringInStrings(stringCollection,
      pattern):
    occ = list()
    for i, s in enumerate(stringCollection):
        j = s.find(pattern)
        while j != -1:
            occ.append((i, j))
            j = s.find(pattern, j+1)
    return occ
```

## myscript.py

```
#!/usr/bin/env python3

from mystringutils import
    findSubstringInStrings

if __name__ == '__main__':
    myStringList = ['the rain in spain',
        'ain\'t no sunshine',
        'she was greeted with disdain']

    occOfAin = findSubstringInStrings(
        myStringList, 'ain')
    print(occOfAin)
```

# Packages

- Way of structuring multiple modules into a directory hierarchy
- Package directories must contain a `__init__.py` file
- Can be imported the same way as modules
- Python itself offers many packages, and even more third-party packages are available through *package managers* such as `conda`

# Quiz

- In Python, a class is _____ for an object.

  a nuisance  an instance  a blueprint  a distraction

- Consider the following class:

```
1 class Dog:
2     def __init__(self, name, age):
3         self.name = name
4         self.age = age
```

What is the correct statement to instantiate a `Dog` object?

- `Dog('Rufus', 3)`
- `Dog(self, 'Rufus', 3)`
- `Dog.__init__('Rufus', 3)`

source (in part): https://realpython.com/quizzes

# Quiz

- In Python, a class is _____ for an object.

  a nuisance  an instance  a blueprint ✔  a distraction

- Consider the following class:

```
1 class Dog:
2     def __init__(self, name, age):
3         self.name = name
4         self.age = age
```

What is the correct statement to instantiate a `Dog` object?

- `Dog('Rufus', 3)` ✔
- `Dog(self, 'Rufus', 3)`
- `Dog.__init__('Rufus', 3)`

source (in part): https://realpython.com/quizzes

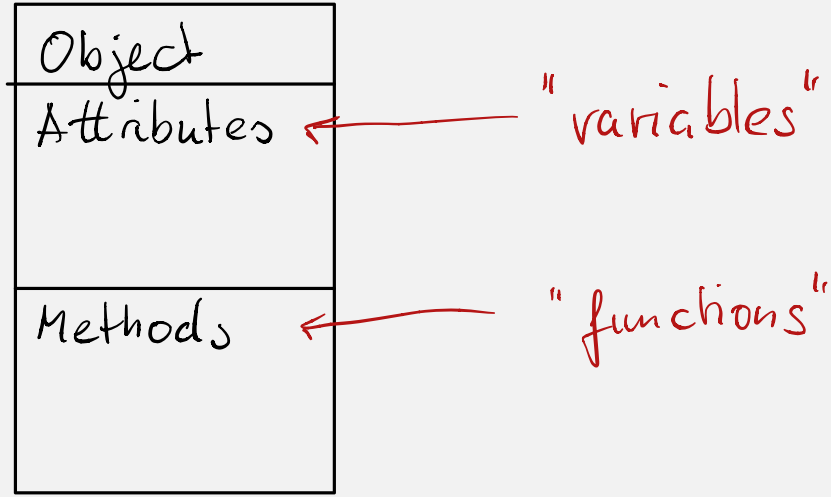
**Functional Programming**

**Lazy Evaluation**

**Object-oriented Programming**

UNIVERSITÄT BIELEFELD

# What is an object?




An object is an instance of a class

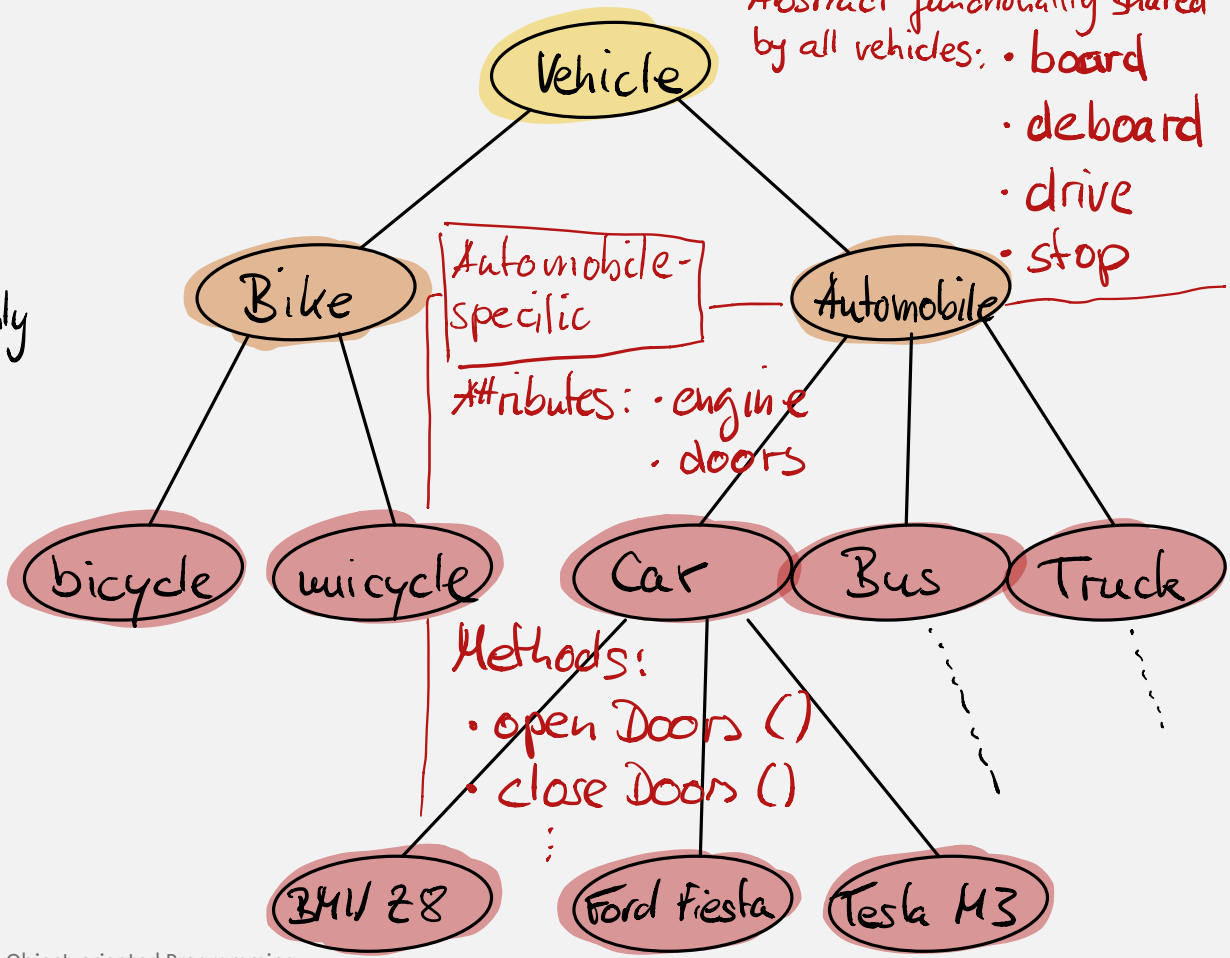
instance

blueprint

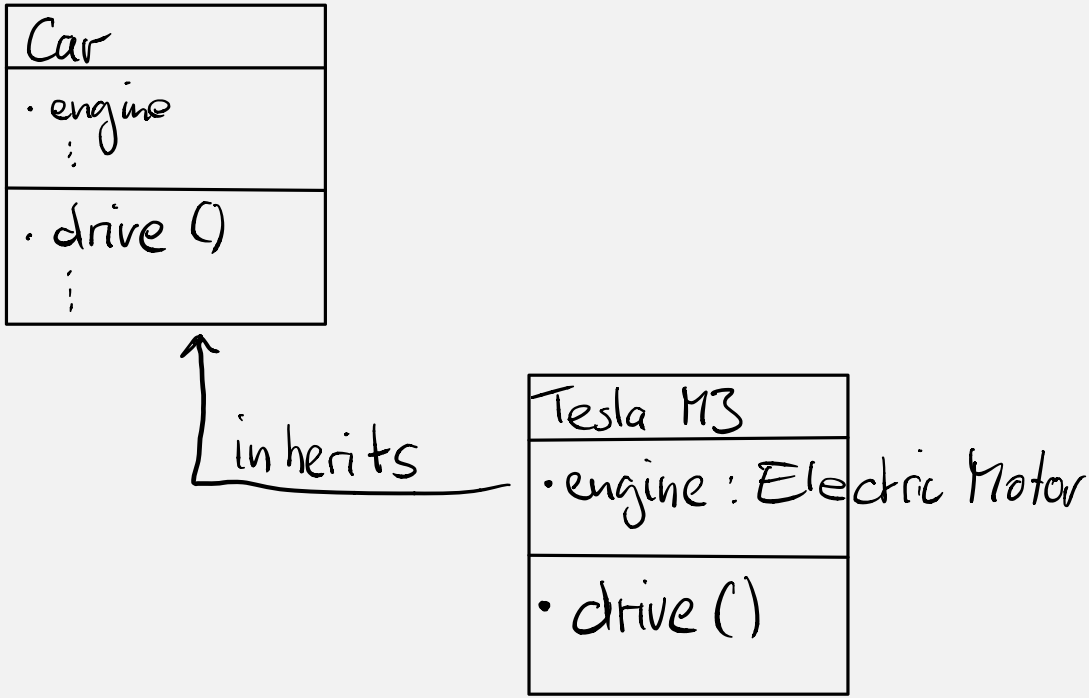
UNIVERSITÄT BIELEFELD

# Inheritance

- **Interface**
  - no impl.
  - typically methods only
- **Abstract Class**
  - partial impl.
- **Class**
  - specific implement.



# Overwriting methods and attributes



# Design principles of software development

## SOLID

- **S**ingle responsibility principle: a class should have only a single responsibility
- **O**pen/closed principle: “software entities [..] should be open for extension, but closed for modification”
- **L**iskov substitution principle: “objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program”
- **I**nterface segregation principle: “many client-specific interfaces are better than one general-purpose interface”
- **D**ependency inversion principle: one should “depend upon abstractions, [not] concretions”

source: `https://en.wikipedia.org/wiki/SOLID`

# Naming conventions

Methods/attributes of the type:

- `some_name` or `someName`: *public*
- `_some_name` or `_someName`: *weak internal use*
- `__some_name` or `__someName`: *strong internal use*
- `__some_name__`: *Python "magic" attribute/function*

Variable named _, e.g.

```
1 for _ in range(10):
2     ...
```

... indicates that it will never be used

# Object-oriented Programming

Every class that you create will be inherited from the `object` class, even if you don't specify explicity, as done in this example.

In [31]:
```
class MyObject(object):
    pass

', '.join(dir(MyObject))
```

Out[31]:
```
'__class__, __delattr__, __dict__, __dir__, __doc__, __eq__, __format__, __ge_
_, __getattribute__, __gt__, __hash__, __init__, __init_subclass__, __le__, __
lt__, __module__, __ne__, __new__, __reduce__, __reduce_ex__, __repr__, __seta
ttr__, __sizeof__, __str__, __subclasshook__, __weakref__'
```

Overwriting inherited methods is simple:

In [32]:
```
class MyObject2(object):
    def __str__(self):
        return 'It\'s my object!'

myObj  = MyObject()
myObj2 = MyObject2()

str(myObj), str(myObj2)
```

Out[32]:
```
('<__main__.MyObject object at 0x10a1bc1d0>', "It's my object!")
```

In [33]:
```
print(myObj2)
```

```
It's my object!
```

## Inheritance

This example showcases the use of interfaces, abstract classes, and (ordinary) classes.

The `Vehicle` interface:

In [34]:
```
class Vehicle:

    def board(self, driver):
        raise NotImplementedError()

    def deboard(self):
        raise NotImplementedError()

    def drive(self):
        raise NotImplementedError()

    def stop(self):
        raise NotImplementedError()
```

Abstract class `Automobile` provides a partial implementation of `Vehicle`:

In [35]:
```
class Automobile(Vehicle):

    def __init__(self, name):
        self.name = name
        self.doors = 'generic doors'
        self.driver = None
        self.engine = None

    def board(self, driver):
        if self.driver != None:
            raise Exception('This automobile is already boarded!')
        self.openDoors()
        print(f'seating driver {driver}')
        self.driver = driver
        self.closeDoors()

    def deboard(self):
        if self.driver == None:
            raise Exception('This automobile is not boarded!')
        self.openDoors()
        print(f'deboarding driver {self.driver}')
        self.driver = None
        self.closeDoors()

    def openDoors(self):
        print(f'opening {self.doors}')

    def closeDoors(self):
        print(f'closing {self.doors}')
```

Example of an "implemented" class, ready to be be instantiated:

In [36]:
```
class Engine:
    def start(self):
        print(f'starting {self}')

    def stop(self):
        print(f'stopping {self}')

class Car(Automobile):

    def __init__(self, name, engine):
        super().__init__(name)
        self.engine = engine

    def drive(self):
        if self.driver == None:
            raise Exception('Car has no driver!')
        self.engine.start()
        print(f'driving forward')

    def stop(self):
        print('hitting breaks')
        self.engine.stop()
```

Derivations of the class, that extend the `Car` class by specific implementations:

In [37]:
```
class ElectricEngine(Engine):
    pass

class TeslaM3(Car):

    def __init__(self):
        super().__init__('Tesla M3', ElectricEngine())

    def drive(self):
        if self.driver == None:
            print('setting autonomous driving mode')
            self.board('Autonomous Driver')
        self.engine.start()
        print(f'driving forward')
```

In [38]:
```
my_tesla = TeslaM3()
my_tesla
```

Out[38]:
```
<__main__.TeslaM3 at 0x10a19f510>
```

In [39]:
```
my_tesla.name
```

Out[39]:
```
'Tesla M3'
```

In [40]:
```
my_tesla.board('Elon Musk')
```

```
opening generic doors
seating driver Elon Musk
closing generic doors
```

In [41]:
```
my_tesla.deboard()
```

```
opening generic doors
deboarding driver Elon Musk
closing generic doors
```

In [42]:
```
my_tesla.drive()
```

```
setting autonomous driving mode
opening generic doors
seating driver Autonomous Driver
closing generic doors
starting <__main__.ElectricEngine object at 0x10a19fcd0>
driving forward
```

In [43]:
```
my_tesla.stop()
```

```
hitting breaks
stopping <__main__.ElectricEngine object at 0x10a19fcd0>
```

In plain Python, inheritance from explicit interfaces is not necessary. Functionality of objects is defined merely by presence of the corresponding functions. Here is an example of the "Iterator" interface:

In [44]:
```
class RepeatIterator:
    def __init__(self, repetitions, value):
        """ Constructor: requires repetitions (integer) and the value that will be
        repeated """
        self.counter = repetitions
        self.val = value

    def __iter__(self):
        """ Implementation of the Iter interface, returns object itself."""
        return self

    def __next__(self):
        """ Will return the repeated element as long as the number of repetitions
        is not exceeded. """
        if self.counter > 0:
            self.counter -= 1
            return self.val

        raise StopIteration
```

In [45]:
```
myIt = RepeatIterator(10, 'Hello World')

print(myIt)

for x in myIt:
    print(x)
```

```
<__main__.RepeatIterator object at 0x10a1cb810>
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
```

In [46]:
```
myIt = RepeatIterator(10, 'Hello World')

next(myIt)
```

Out[46]:
```
'Hello World'
```

Functions are objects that implement a `__call__` function:

In [47]:
```
class MyCallable:

    def __call__(self, *args):
        """ Implementation of the Call interface, returns passed on parameters"""
        return args

myCall = MyCallable()

print(myCall)
```

```
<__main__.MyCallable object at 0x10a1cec90>
```

In [48]:
```
myCall('Hello', 1, 2, 3)
```

Out[48]:
```
('Hello', 1, 2, 3)
```

# Quiz

- *True* or *false?*
  - Every function in Python is an object
  - Methods of parent class cannot be overridden
  - All classes are derived from the same base class
  - Classes inherit all variables and functions from the parent class
  - Python's "magic" functions can't be overridden.
- Order the variable names by increasing privacy.
  - `_some_name`
  - `some_name`
  - `__some_name`

# Quiz

- *True* or *false?*
  - Every function in Python is an object — true
  - Methods of parent class cannot be overridden — false
  - All classes are derived from the same base class — true
  - Classes inherit all variables and functions from the parent class — true
  - Python's "magic" functions can't be overridden. — false
- Order the variable names by increasing privacy.
  - `_some_name` — 2.
  - `some_name` — 1.
  - `__some_name` — 3.