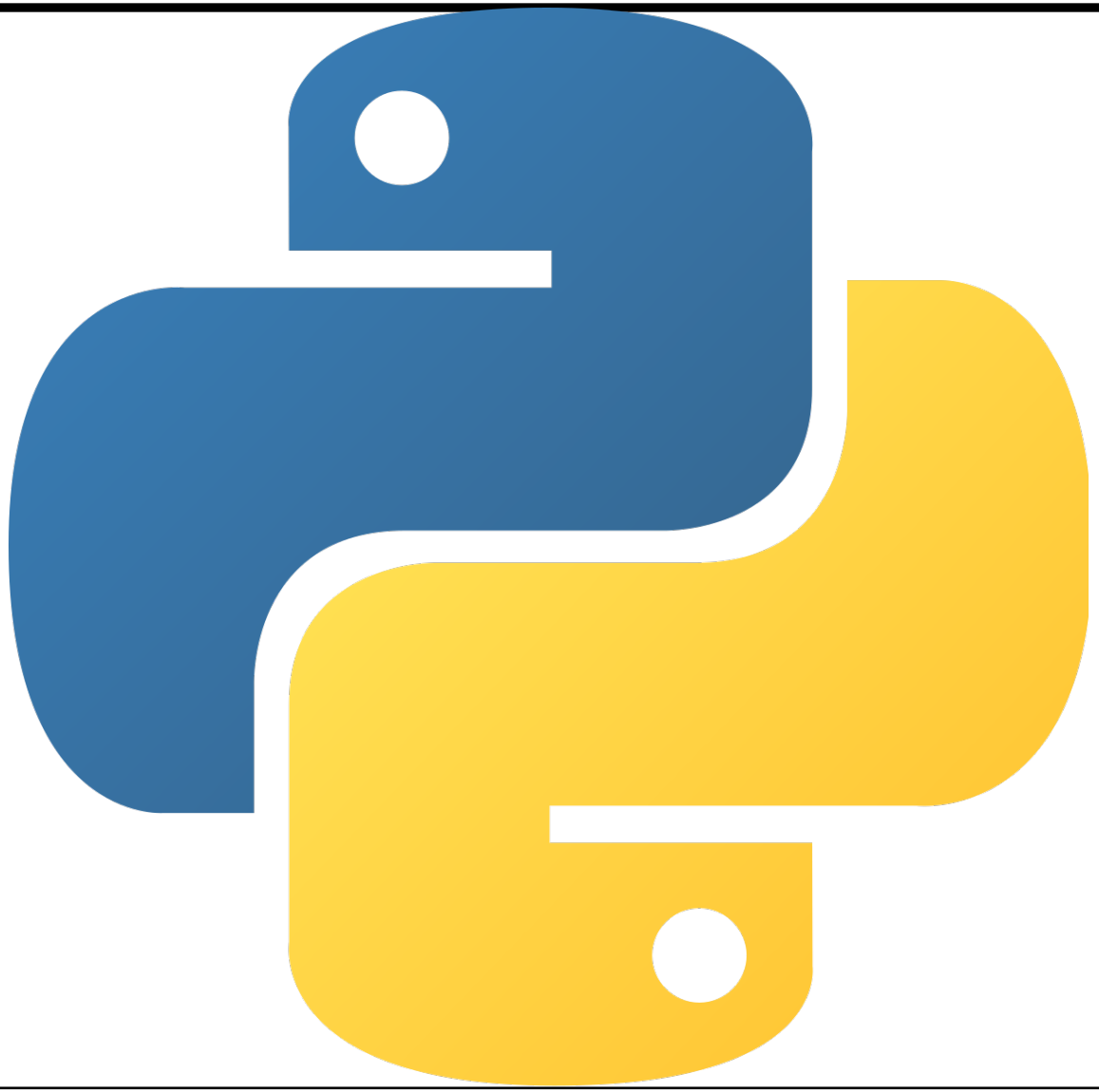
PYTHON PROGRAMMERING



Föreläsning 9

DAGENS FRÅGA

- Vad behöver moderniseras?

DAGENS AGENDA

- Inlämningsuppgift
- Mentimeter – mittpunkt
- **Objektorienterad programmering OOP:**
- Grundpelarna i OOP
- Klasser
- Privata och publika metoder
- Static method
- Arv och superklass

FÖRRA FÖRELÄSNING

- Pathlib – bättre hantering av path
- Pandas:
  - Series
  - Dataframes
  - Index: iloc() och loc()

INLÄMNINGSUPPGIFT

MENTIMETER – MITTPUNKTSVÄRDERING

6944 4039

BAKGRUNG

- Hittills har vi skrivit python procuderellt, alltså definierat variabler och skrivit egna funktioner
- När man skriver större program blir det snabbt ineffektivt
- Vi vill ha kod som...
  - går att återanvända
  - är organiserad och lätt att förstå
  - är lätt att modifiera och felsöka

- Lösningen: **Objektorienterad programmering**
- https://www.youtube.com/watch?v=pTB0EiLXUC8&ab_channel=ProgrammingwithMosh 7 min video

Class

Object

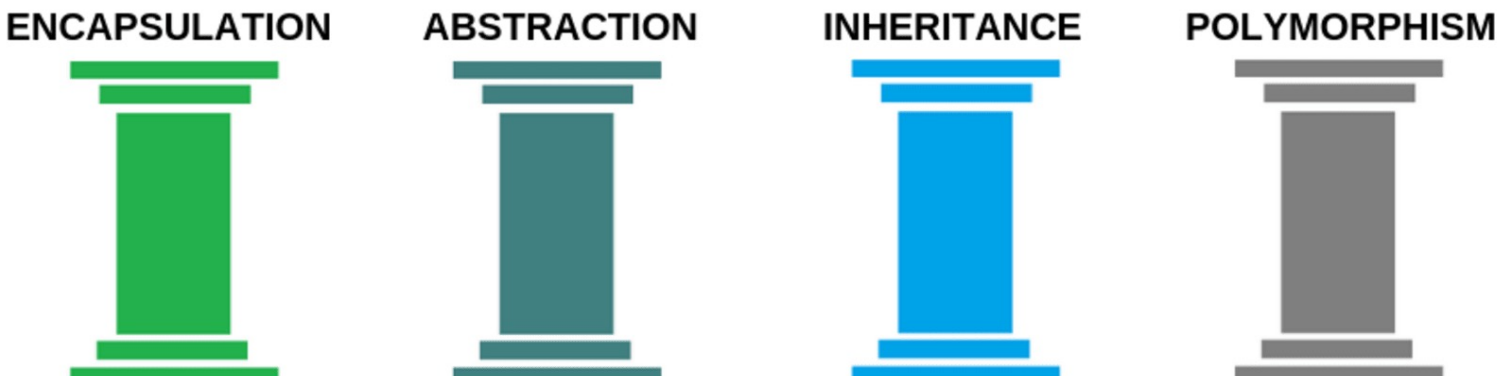
Pokemon
Name: Pikachu
Type: Electric
Health: 70
attack()
dodge()
evolve()

Attributes

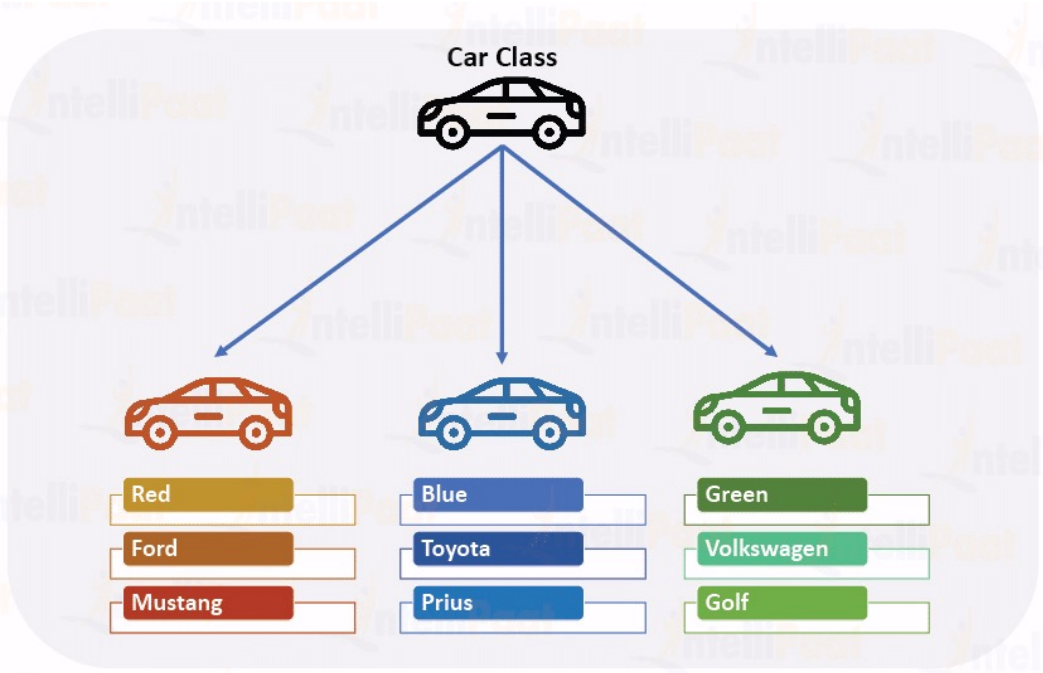
Methods

GRUNDPELARNA I OOP

- **Abstraktion:** att simplifiera användning genom att gömma implementering → interface
- **Enkapsulering:** att samla funktionalitet och data på ett ställe → classes
- **Arv (Inheritance):** att kunna återanvända gemensam funktionalitet → parent classes
- **Polymorfism:** att kunna ändra beteendet av gemensam funktionalitet → overriding



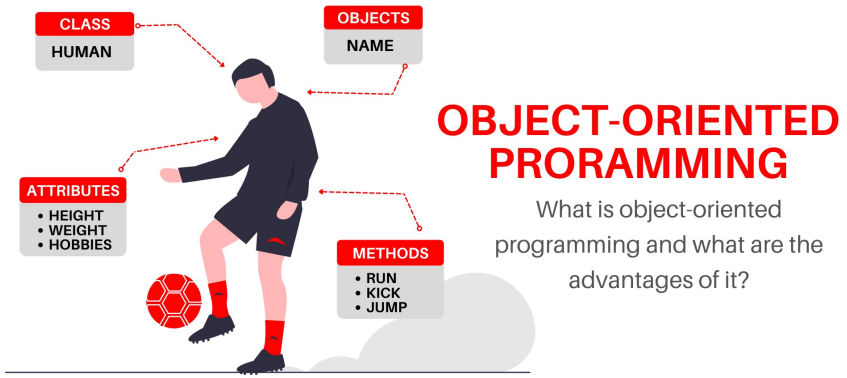
KLASSER



- I vissa fall är det användbart att kunna gruppera data och funktioner som hör ihop. Detta görs med hjälp av **klasser/classes**
- En klass är en sorts mall eller definition av ett objekt som definierar dess egenskaper
  - I python är alla entiteter objekt, även enkla datatyper som int och float
- *Inom data science är det inte ovanligt att OOP läggs åt sidan. Men att skriva objektorienterat kan ibland vara extremt effektivt i längden när kod ska återanvändas*
- *I princip alla pythonbibliotek ni interagerar med använder klasser på olika sätt*

KLASSER - TERMINOLOGI

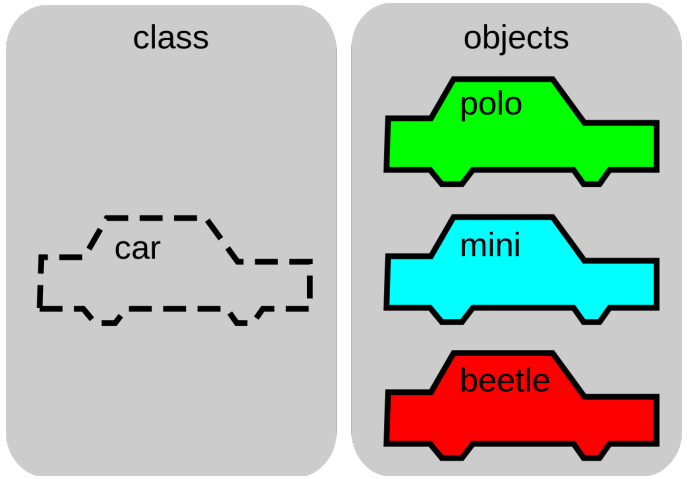
- **Attributes**: den data som sparas i ett objekt
- **Methods**: de funktioner som associeras med ett objekt
- **self**: en referens till ett objekt i dess egna metoders definitioner
- **Initialiser**: den metod som kallas när ett objekt skapas
- **Instance**: en specifik realisering av en klass




```
class Thing():

    def __init__(self, param1, param2):
        """ This is the initializer of the class """
        self.attribute1 = param1
        self.attribute2 = param2
```

OBJECTS

- För att skapa ett nytt objekt av en klass kallar man på klassen och tilldelar en variabel denna (på samma sätt vi hade gjort med funktioner)
- Variablerna till detta objektet delas inte mellan andra objekt av samma klass (utom klass/static varaibler)
- Varje objekt kan ses som en beskrivning av sin egen variant av klassen




```
class Person: # create a class with name person
    first_name = "Foo" # class variable

foo = Person() # Create a Person object and assign it to the foo variable
bar = Person() # Create a Person object and assign it to the bar variable
print(foo.first_name) #prints Foo
print(bar.first_name) #prints Foo
bar.first_name = "Bar" # changing first_name of this Person Instace
print(foo.first_name) #prints Foo (no change)
print(bar.first_name) #prints Bar (changed)
```

CLASS VARIABLES

- Klassvariabel betyder att klassen äger variabeln – den finns i klassen – så alla instanser av klassen kommer att kunna komma åt den variabeln.
- Klassvariabler delas av alla instanser som har åtkomst till klassen
- Dessa statiska variabler kan nås utan att skapa en instans av objektet
- Enligt konvention placeras dessa direkt under klassen och ovanför konstruktorn

```
class Person: # create a class with name person
    first_name = "Foo" # class variable

foo = Person() # Create a Person object and assign it to the foo variable
bar = Person() # Create a Person object and assign it to the bar variable
print(foo.first_name) #prints Foo
print(bar.first_name) #prints Foo
bar.first_name = "Bar" # changing first_name of this Person Instace
print(foo.first_name) #prints Foo (no change)
print(bar.first_name) #prints Bar (changed)
```

SELF

```python
class User:
    def __init__(self, id, name = ""): # ctor taking two parameters
        self.id = id # instance variable
        self.name = name # instance variable
        self.test = "Test" # instance variable

    def change_name(self, new_name): # class methods that take one parameter
        self.name = new_name # access instance variables using self

foo = User(1, "Foo")
bar = User(2) # Name is an optional parameter
print(foo.id, foo.name, foo.test) # Prints 1 Foo Test
print(bar.id, bar.name, bar.test) # Prints 2 Test
bar.change_name("Bar") # call function change_name for bar instance of User
print(bar.name) # Prints Bar
```

- För att komma åt variabler och metoder från klassen måste vi använda nyckelordet ***self***
- *Self* håller en referens till sig själv
- Vad man gör är att instruera Python för att få min egen (*self*) kopia av variabeln
- Alla metoder som definieras i en klass måste använda nyckelordet *self* som första parameter för att kunna komma åt objektets instansvariabler

INSTANCE VARIABLER

```python
class User:
    def __init__(self, id, name = ""): # ctor taking two parameters
        self.id = id # instance variable
        self.name = name # instance variable
        self.test = "Test" # instance variable

    def change_name(self, new_name): # class methods that take one parameter
        self.name = new_name # access instance variables using self

foo = User(1, "Foo")
bar = User(2) # Name is an optional parameter
print(foo.id, foo.name, foo.test) # Prints 1 Foo Test
print(bar.id, bar.name, bar.test) # Prints 2 Test
bar.change_name("Bar") # call function change_name for bar instance of User
print(bar.name) # Prints Bar
```

- Instansvariabler är unika default värden för varje instans av en klass.
- Instansvariabler definieras med self.variable_name i konstruktorn *def __init__(self, name)*

CONSTRUCTORS, METHODS

```python
class User:
    def __init__(self, id, name = ""): # ctor taking two parameters
        self.id = id # instance variable
        self.name = name # instance variable
        self.test = "Test" # instance variable

    def change_name(self, new_name): # class methods that take one parameter
        self.name = new_name # access instance variables using self

foo = User(1, "Foo")
bar = User(2) # Name is an optional parameter
print(foo.id, foo.name, foo.test) # Prints 1 Foo Test
print(bar.id, bar.name, bar.test) # Prints 2 Test
bar.change_name("Bar") # call function change_name for bar instance of User
print(bar.name) # Prints Bar
```

- Constructors är en funktion med namnet *__init__(self, name)*
- Constructors anropas automatisk innan klassen skapas
- Den bör användas för att konfigurera allt som krävs för att klassen ska fungera
- Funktioner i klassen kallas ***metoder*** och skapas med ***def***
- Första parametern i en metod är *self*
- När man anropar metoder måste man inte skriva *self*, Python gör det automatisk

PRIVATE/PUBLIC METHODS

```python
class Thing():

    def __init__(self, param1, param2):
        """ This is the constructor of the class """
        self.attribute1 = param1
        self.attribute2 = param2

    def public_func(self):

        return

    def _private_func(self):

        return
```

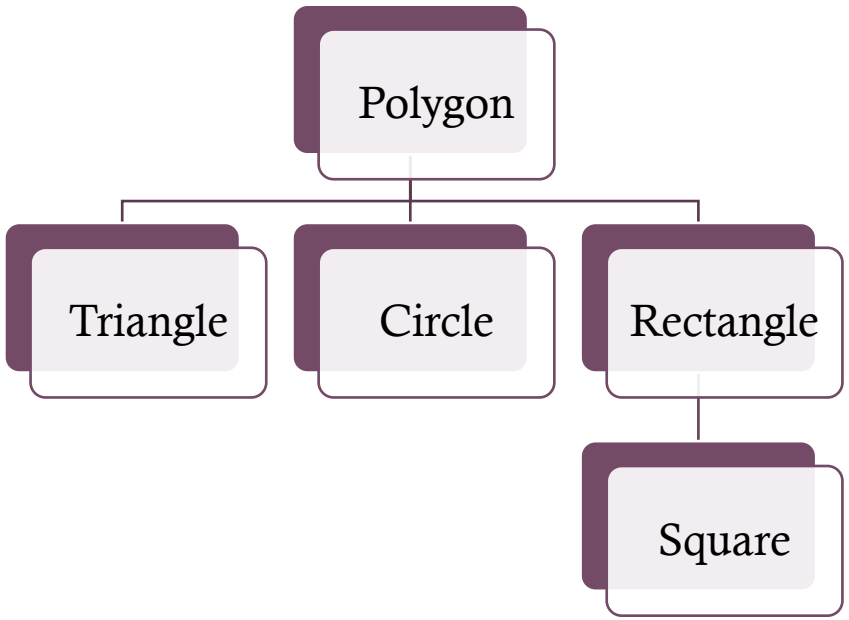
- I andra programmeringsspråk är det vanligt med publika och privata metoder. Detta är inte implementerat i python
  - Public method: ska användas utanför en objektinstans
  - Private method: ska bara användas inom objektet och är oftast hjälpmetoder
- I python är konventionen att markera privata metoder med ett underscore för att berätta för en användare att de inte är lämpliga att använda utanför klassen

```python
class Number():

    def __init__(self, value):
        self.value = value


    @staticmethod
    """ This is a method that does not require an instance of the class to be executed"""
    def static_sum(value1, value2):
        return value1 + value2
```

THE STATIC METHOD

- Ibland stöter man på decoratorn @staticmethod
- En statisk metod kräver inte att en instans av klassen finns och tar därför inte *self* som parameter i metodens definition
- Kan t.ex. användas som en alternativ initialiser eller för en funktion som verkligen knyter an till klassen

ARV OCH SUPERKLASS



- I OOP är arv ett sätt att skapa en hierarki av objekt och klasser
- En klass kan *ärva* från en *super class / parent class* vilket innebär att den har tillgång till samma metoder och attribut
- En subklass kan skugga/override en metod den ärver från sin parent class
- För att använda arv på ett strukturerat sätt så låter man mer specifika klasser ärva från mer generella