Lær Python dag 4 - modul 1

Klasser og objekter

Jonas Bamse Andersen

Institut for Matematik og Datalogi - IMADA
Syddansk Universitet

Indhold

1. Recap
2. Objektorienteret programmering
3. Klasser og objekter i python
4. Metoder
5. Static methods
6. Magic methods m.m.
7. Hvad har I lært?

Recap

Recap

Hvad lærte I sidst? Diskutér med sidemanden 2 min.

- Hvad er et dictionary? Hvad kan det bruges til?
- Hvordan interagerer dictionaries og loops?
- Hvordan bruger vi filer i Python?
- Hvordan interagerer filer og loops?
- Hvad er `try` og `except`? Hvad kan det bruges til?
- Andet?

Objektorienteret programmering

Objektorienteret programmering

Termet Objektorienteret Programmering (OOP) har i mange år været genstand for en del diskussion. Jeg forklarer her, hvad jeg mener er den mest udbredte og naturlige anvendelse af konceptet.

Objektorienteret programmering

Termet Objektorienteret Programmering (OOP) har i mange år været genstand for en del diskussion. Jeg forklarer her, hvad jeg mener er den mest udbredte og naturlige anvendelse af konceptet.

OOP er en måde at programmere og strukturere sin kode på.

Objektorienteret programmering

Termet Objektorienteret Programmering (OOP) har i mange år været genstand for en del diskussion. Jeg forklarer her, hvad jeg mener er den mest udbredte og naturlige anvendelse af konceptet.

OOP er en måde at programmere og strukturere sin kode på.

Som navnet antyder handler OOP om objekter. Objekter modellerer ofte virkelige ting som f.eks. en bil, men kan også bruges til at modellere mere abstrakte ting som f.eks. en forbindelse til en database.

Objektorienteret programmering

Grundlæggende er der to ting som et objekt består af:

- En samling værdier (attributes/fields)
- En samling funktioner, der kan manipulere objektet (methods).

Objektorienteret programmering

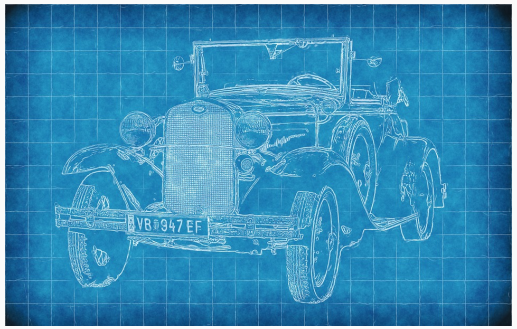
Grundlæggende er der to ting som et objekt består af:

- En samling værdier (attributes/fields)
- En samling funktioner, der kan manipulere objektet (methods).

Objekter er defineret udfra en såkaldt klasse.

Klasser og objekter

Klasse: Et "blueprint" der beskriver, hvad en ting (objekt) af denne type består af og kan.



Objekt: En unik instans af en klasse, som kan manipuleres.

Eksempler på klasser

- Bil:
  Attributes: antal døre, producent, model
  Metoder: dyt(), stop(), start()
- Person:
  Attributes: navn, alder, forældre (Person)
  Metoder: hils(), skift_navn(), løb()

Har vi allerede brugt objekter?

Har vi allerede brugt objekter?

Vi har fx set et filobjekt:

```
f = open("min_fil.txt", "w")
f.write("Lær python!!!\n")
f.write("IMADA SDU")
f.close()
```

Her bruger vi metoderne `write()` og `close()` på vores filobjekt.

Har vi allerede brugt objekter?

Filobjektet har også attributes fx:

- name: Navnet på filen (string)
- closed: Er filen lukket eller ej (boolean)
- mode: Hvordan er filen blivet åbnet (string)

```
f = open("min_fil.txt", "w")
print(f.name)
print(f.closed)
print(f.mode)
f.close()
```

```
min_fil.txt
False
w
```

Klasser og objekter i python

Definition af klasse

En klasse defineres på følgende vis:

```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age
```

`__init__` er en speciel metode som bliver kaldt når et nyt objekt skal laves og kaldes for en *constructor* metode. Denne metode skal have `self` som første parameter. Dette er en reference til objektet selv og bruges til at få adgang til objektets attributes.

Initialisering

Et nyt objekt kan nu laves, ved at :

```
person1 = Person("Bamse", 25)
person2 = Person("Ruth", 78)
```

Og attributter kan tilgås vha. .-operatoren:

```
print("person1:", person1.name, person1.age)
print("person2:", person2.name, person2.age)
```

Output:

```
person1: Bamse 25
person2: Ruth 78
```

Hvor kan de bruges?

Objekter kan bruges alle steder hvor I er vant til at bruge variabler, f.eks.

- Som parameter til en funktion.
- Som en attribut i et andet objekt.
- Som returværdi fra en funktion.

Eksempel

Lad os lave en klasse som gemmer et klokkeslæt:

```
class Time:
  def __init__(self, h, m, s):
      self.hours = h
      self.mins = m
      self.secs = s
```

Eksempel

Lad os lave en klasse som gemmer et klokkeslæt:

```
class Time:
  def __init__(self, h, m, s):
      self.hours = h
      self.mins = m
      self.secs = s
```

Vi kan lave en funktion som printer tiden:

```
def print_time(t):
  print("%02d:%02d:%02d" % (t.hours, t.mins, t.secs))
```

Eksempel

Lad os teste det:

```
t = Time(2, 3, 23)
print_time(t)
```

```
02:03:23
```

Metoder

Hvad er det nye?

Måske undrer I jer over, hvorfor man skulle bruge objekter, når man kunne gøre det samme med et dictionary:

```
def make_time(h, m, s):
    return {"hours": h,
            "mins": m,
            "secs": s}

def print_time(t):
    print("%02d:%02d:%02d" % (t["hours"], t["mins"], t["secs"]))

t = make_time(13,37,42)
print_time(t)
```

Hvad er det nye?

Et problem kunne være hvis man ikke giver funktionen det rigtige:

```
print_time ("hej")
```

Hvad er det nye?

Et problem kunne være hvis man ikke giver funktionen det rigtige:

```
print_time("hej")
```

```
Traceback (most recent ...
File "test.py", line 6...
print_time("hej")
File "test.py", line 2...
print("%02d:%02d:%02d"...
TypeError: string indices
    must be integers
```

Hvad er det nye?

Vi kan løse det med metoder. Metoder er funktioner der tilhører objekter. Metoder skrives som en del af en klassedefinition.

```
class Time:
    ...

    def print_time(self):
        print("%02d:%02d:%02d" % (self.hours, self.mins, self.secs
            ))
```

Hvad er det nye?

Vi kan løse det med metoder. Metoder er funktioner der tilhører objekter. Metoder skrives som en del af en klassedefinition.

```
class Time:
    ...

    def print_time(self):
        print("%02d:%02d:%02d" % (self.hours, self.mins, self.secs
            ))
```

Vi har flyttet *ansvaret* for at printe, til objektet selv.

HUSK metoder skal altid have `self` som første argument!

Desuden må ens metode ikke hedde det samme som en attribut. (f.eks. "sekunder"i dette tilfælde)

Hvad er det nye?

Når man skal bruge metoden bruger man punktum.

```
...
t = Time(13,37,42)
t.print_time()
```

Hvad er det nye?

Når man skal bruge metoden bruger man punktum.

```
...
t = Time(13,37,42)
t.print_time()
```

```
13:37:42
```

Hvad er det nye?

Når man skal bruge metoden bruger man punktum.

```
...
t = Time(13,37,42)
t.print_time()
```

```
13:37:42
```

Som I kan se, kan man nu ikke komme til at kalde funktionen forkert.

Tiden går

Vi kan også ændre på et objekts værdier i en metode.

```
class Time:
    ...
    def wait(self, sekunder):
        self.secs += sekunder

        self.mins += self.secs // 60
        self.secs = self.secs % 60

        self.hours += self.mins // 60
        self.mins = self.mins % 60

        self.hours = self.hours % 24
```

Tiden går

Nu kan vi se, hvordan tiden går:

```
...
t = Time(13,37,42)
t.print_time()
t.wait(59)
t.print_time()
```

Tiden går

Nu kan vi se, hvordan tiden går:

```
...
t = Time(13,37,42)
t.print_time()
t.wait(59)
t.print_time()
```

```
13:37:42
13:38:41
```

Eksempel

Husk stadig, at variabel navne er referencer til objekter, så et objekt kan blive ændret "et andet sted fra".

```
t1 = Time(1, 2, 3)
t2 = t1
t2.print_time()
t1.wait(1)
t2.print_time()
```

```
01:01:01
01:01:02
```

Static methods

Static methods

Nogle gange vil man gerne associere bestemte funktioner med en bestemt klasse, uden at de er knyttet til et specifikt objekt. I de tilfælde kan man lave static methods.

Static methods

Nogle gange vil man gerne associere bestemte funktioner med en bestemt klasse, uden at de er knyttet til et specifikt objekt. I de tilfælde kan man lave static methods.

I vores tidseksempel kunne det være man vil have en funktion der altid kan give os frokost tidspunktet 12:00:00. Det kunne se således ud:

```
class Time:
    ...

  def noon():
    return Time(12,0,0)

t = Time.noon()
t.print_time()
```

```
12:00:00
```

Static methods

Nogle gange vil man gerne associere bestemte funktioner med en bestemt klasse, uden at de er knyttet til et specifikt objekt. I de tilfælde kan man lave static methods.

I vores tidseksempel kunne det være man vil have en funktion der altid kan give os frokost tidspunktet 12:00:00. Det kunne se således ud:

```
class Time:
    ...

  def noon():
    return Time(12,0,0)

t = Time.noon()
t.print_time()
```

```
12:00:00
```

Static methods

Et andet eksempel kunne være at vi ville have en funktion der kan sammenligne to personer, og svare på om den ene er ældre end den anden.

```
class Person:
    ...

    def older(p1, p2):
        return p1.age > p2.age

jens = Person("Jens", 35)
peter = Person("Peter", 27)
print(Person.older(jens, peter))
```

Static methods

Et andet eksempel kunne være at vi ville have en funktion der kan sammenligne to personer, og svare på om den ene er ældre end den anden.

```
class Person:
    ...

    def older(p1, p2):
        return p1.age > p2.age

jens = Person("Jens", 35)
peter = Person("Peter", 27)
print(Person.older(jens, peter))
```

```
True
```

Magic methods m.m.

Specielle metoder

Vores tidligere metode `print_time` hjælper os med at printe tiden, men hvad nu hvis man gerne vil printe den sammen med noget andet?

```
Klokken er 13:37:42 lige nu.
```

Specielle metoder

Vores tidligere metode `print_time` hjælper os med at printe tiden, men hvad nu hvis man gerne vil printe den sammen med noget andet?

```
Klokken er 13:37:42 lige nu.
```

Og hvad sker der egentlig hvis vi printer tid, lige nu?

```
...
t = Time(13,37,42)
print(t)
```

Specielle metoder

Vores tidligere metode `print_time` hjælper os med at printe tiden, men hvad nu hvis man gerne vil printe den sammen med noget andet?

```
Klokken er 13:37:42 lige nu.
```

Og hvad sker der egentlig hvis vi printer tid, lige nu?

```
...
t = Time(13,37,42)
print(t)
```

```
<__main__.Time object at
      0x02B11890>
```

Specielle metoder

Der findes en masse specielle metoder, nogle gange kaldet magic methods, som bliver kaldt på nogle bestemte tidspunkter og har nogle bestemte krav. Man kan kende dem på at de har to underscores både før og efter navnet, så I har allerede set én, `__init__`, som kaldes for en constructor funktion.

En anden meget brugt af disse er `__repr__` som skal returnere en streng, og meningen er at det er sådan objektet repræsenteres som en streng.

```
class Time
  ...
  def __repr__(self):
    return "%02d:%02d:%02d" % (self.hours, self.mins, self.
        secs)
```

Specielle metoder

Så nu behøver vi ikke `print_time` metoden længere, og vi har fået endnu mere fleksibilitet med.

```
...
t = Time(13,37,42)
print(t)
print("Klokken er",
      t, "lige nu.")
```

Specielle metoder

Så nu behøver vi ikke `print_time` metoden længere, og vi har fået endnu mere fleksibilitet med.

```
...
t = Time(13,37,42)
print(t)
print("Klokken er",
      t, " lige nu.")
```

```
13:37:42
Klokken er 13:37:42 lige nu.
```

Specielle metoder

Der findes rigtig mange flere, men dem må I selv læse mere om på nettet. Bare husk at de kan række dybt ind i Pythons maskineri, så brug dem med omtanke.

Hvad har I lært?

Hvad har I lært

- Typer
- Variabler
- If-sætninger
- Funktioner
- Løkker
- Lister
- Strenge
- Dictionaries
- Filer
- Klasser & objekter
- Og meget mere

Hvad har I lært

- Typer
- Variabler
- If-sætninger
- Funktioner
- Løkker
- Lister
- Strenge
- Dictionaries
- Filer
- Klasser & objekter
- Og meget mere

Så selv hvis I stadig har svært ved nogle af opgaverne, så kan I gå tilbage og prøve nogle af de første opgaver igen, og mærke hvor meget lettere de er nu