Flik 8: Klasser

Objektorientering allmänt

- Ett *objekt* är ett stycke data
  - ofta sammansatt av flera delar
- En *klass* är ritningen/definitionen av ett objekt
  - Definierar vad man kan göra med objektet
- Ett nytt objekt är en realisering utifrån denna klass, t.ex.:
  - Ritning/fabrik/tillverkningsprocess = Klass
  - Varje ny Volvo som åker ut ur fabriken är ett nytt objekt (som kan ha unika egenskaper som modell, färg osv.)

```
>>> min_bil = Volvo('S80','Blå','XZU243',....)
```

- En klass kan ärva egenskaper från en annan klass, t.ex.:
  - Fordon → Bil → Volvo

Skapa en instans

```
>>> class Datum:                    # Tom klass
...     pass

>>> idag = Datum()                  # Ny instans
>>> idag.y = 2019
>>> idag.m = 10
>>> idag.d = 9

>>> julafton = Datum()              # Ny instans
>>> julafton.y = 2019
>>> julafton.m = 12
>>> julafton.d = 24
```

- En klass kan som i detta fall vara minimal
- Instansattribut kan skapas dynamiskt (påverkar inte klassen)
- Vanligare: **Använd metoder för att läsa/skriva attribut**
  - datainkapsling

Metoder

```
class Datum:
    def skriv_ut(self):
        s = '{0:d}-{1:02d}-{2:02d}'
        print(s.format(self.y, self.m, self.d))

idag = Datum()

idag.y = 2019
idag.m = 10
idag.d = 9

idag.skriv_ut()    # Datum.skriv_ut(idag)
```

- Argumentet *self* måste finnas i alla metoddeklarationer
  (statiska metoder och klassmetoder undantagna - mer om det senare)
- Av konvention ska det heta just *self* (heter *this* i t.ex. C++/Java)

Konstruktorn __init__

- Specialmetoden **__*init*__** är en konstruktor, dvs körs när ett objekt skapas

```
class Datum:
    def __init__(self, year, month, day):
        self.y = year
        self.m = month
        self.d = day

    def skriv_ut(self):
        s = '{0:d}-{1:02d}-{2:02d}'
        print(s.format(self.y, self.m, self.d))


>>> idag = Datum(2019, 10, 9)
>>> idag.skriv_ut()
2019-10-09
```

Exempel

```
class Datum:
    def __init__(self, year, month, day):
        self.y, self.m, self.d = year, month, day

    def datum(self):
        s = '{0:d}-{1:02d}-{2:02d}'
        return s.format(self.y, self.m, self.d)

    def plus_månad(self, antal):
        m = self.m + antal
        self.m = m % 12
        self.y += m // 12

>>> idag = Datum(2019, 10, 9)
>>> idag.plus_månad(4)
>>> idag.datum()
2020-02-09
```

Arv: Skapa en subklass

```
class Tidpunkt(Datum):
    def __init__(self, y, m, d, H, M, S):
        Datum.__init__(self, y, m, d)
        self.H, self.M, self.S = H, M, S

    def klockslag(self):
        s = "{} {:02d}:{:02d}:{:02d}"
        return s.format(self.datum(), self.H, self.M, self.S)

>>> nu = Tidpunkt(2019, 10, 9, 16, 7, 28)
>>> nu.datum()
'2019-10-09'
>>> nu.klockslag()
'2019-10-09 16:07:28'
```

- Om ingen konstruktor anges anropas basklassens konstruktor
- Om konstruktor anges anropas *inte* basklassens konstruktor automatiskt
- Metoder ärvs från basklassen. Kan åsidosättas/överlagras (*eng. override*)

Strängrepresentation av ett objekt

```
class Datum:
    def __init__(self, year, month, day):
        self.y, self.m, self.d = year, month, day
    def datum(self):
        s = '{0:d}-{1:02d}-{2:02d}'
        return s.format(self.y, self.m, self.d)
    def __str__(self):
        return self.datum()
    def __repr__(self):
        s = 'Datum({}, {}, {})'
        return s.format(self.y, self.m, self.d)

>>> idag = Datum(2019, 10, 9)
>>> print(idag)
2019-10-09
>>> idag
Datum(2019, 9, 10)
```

Överlagring av operatorer

- Det går att skräddarsy hur ett objekt ska bete sig i olika situationer, t.ex.

```
>>> idag = Datum(2019, 10, 9)
>>> julafton = Datum(2019, 12, 24)
# Vad händer i följande fall?
>>> if idag < julafton:
...     bla_bla_bla

# Lägg till funktioner till klassen Datum:
def __eq__(self, other):
    return self.y == other.y and self.m == other.m and self.d == other.d
def __lt__(self, other):
    return (self.y, self.m, self.d) < (other.y, other.m, other.d)
def __le__(self, other):
    return self < other or self == other
```

- Det finns många andra metoder för operatoröverlagring. Tips: `dir("")`

Klassattribut och statiska metoder

- En ”statisk metod” jobbar inte på något enskilt objekt – har inte `self`

```
class Bankkonto:
    diskonto = 0.01

    @staticmethod
    def räntesats():
        return Bankkonto.diskonto

    @staticmethod
    def ändra_räntesats(värde):
        Bankkonto.diskonto = värde

>>> Bankkonto.räntesats()
0.01
>>> k = Bankkonto()
>>> k.räntesats()
0.01
>>> Bankkonto.ändra_räntesats(0.005)
>>> k.räntesats()
0.005
```

Hålla reda på antalet instanser

- Destruktorn `__del__` körs när objektet upphör att existera

```
class InstTrack:
    count = 0
    def __init__(self):
        InstTrack.count += 1
    def __del__(self):
        InstTrack.count -= 1
    @staticmethod
    def how_many():
        return InstTrack.count

>>> a = InstTrack()
>>> b = InstTrack()
>>> b.how_many()
2
>>> del b
>>> InstTrack.how_many()
1
```

Varning: opålitlig destruktor

- `__del__` anropas inte förrän antal referenser har gått ner till 0
- Ibland körs destruktorn långt senare än man tror/önskar...

```
>>> c = a      # Fortf. är count==1 (c endast referens)
>>> del a      # Fortf. är count==1 (c ref till obj)
```

Privata attribut eller metoder

- För att ”dölja” attribut i ett objekt, inled namnet med dubbla _
  - Betyder att attributet är **privat**
  - Dvs. det får inte användas utanför klassen
  - Dvs. det **tillhör implementationen**, inte API:et

Bestämma ett objekts förhållande till andra

- *issubclass(C, B) -> bool*
  returnerar sant om *C* är en subklass till B
- *isinstance(object, class-or-type-or-tuple) -> bool*
  returnerar sant om *object* är en instans av en klass eller subklass
- super() - för att leta efter motsvarande metod i föräldrarklasserna

```
class B(A):
    def __init__(self, arg):
        super().__init__(arg)
        ...
```

Övning

- **ovn0801/stack.py** Skapa en klass Stack.
  - Konstruktorn ska skapa en tom stack.
  - Metoden `push(value)` ska lägga till ett element.
  - Metoden `pop()` ska ta bort och returnera senaste elementet.
  - Om man försöker göra `pop` på en tom stack så ska ett exception av typen `TomStack` genereras.
  - Det ska gå att skriva ut stackinnehållet med `print`.
- **ovn0801/tprog.py** Skriv ett program som
  - importerar klassen `Stack`,
  - konstruerar ett tomt `Stack`-objekt
  - i tur och ordning push:ar `"hej"`, `"hopp"`, `1`, `2`, `3` på objektet,
  - i en “evig” loop gör `pop()` på objektet samt skriver ut returvärdet och objektet
  - fångar undantaget `TomStack` som till slut genereras i loopen ovan.