Aufgabe 1

```
1  class Student:
2
3      # Attribut = Eigenschaft = Variable;  Methode = Funktion
4
5      n = 1 # Klassenattribut; nur 1 Wert pro Klasse; Zugriff mit Student.n
6
7      def __init__(self, name, sem): # Konstruktor mit 1+2 Argumenten
8          self.name = name           # name -> name-Attribut
9          self.semester = sem        # sem -> semester-Attribut
10         self.ID = Student.n        # Student.n -> ID-Attribut
11         Student.n += 1             # Erhöhe Student.n um 1 (nächste ID)
12
13     def __str__(self):  # überschreibt die str()-Funktion
14         return '{0.ID}/{0.name}'.format(self) # gibt Textdarstellung zurück
15
16     # Gibt True zurück, wenn die Semesterzahl von self kleiner als die
17     # Semesterzahl von other ist und False sonst.
18     # "other" ist wie "self" nur eine Python-Konvention. Andere Namen
19     # wie in "__lt__(me, you):" sind möglich aber nicht üblich.
20     def __lt__(self, other): # überschreibt Operator "<" ([l]ess [t]han
21         return self.semester < other.semester
22
23     def next_semester(self): # Objektmethode
24         self.semester += 1   # erhöht semester-Eigenschaft eines Objekts
25
26 # Ende der Klassendefinition und Beginn der Klassenanwendung
27
28 # Hier werden 3 Objekte instanziiert (erzeugt), die ID kommt von Student.n
29 s1 = Student('Mary', 3) # Student-Objekt mit name='Mary', semester=3, ID=1
30 s2 = Student('John', 5) # Student-Objekt mit name='John', semester=5, ID=2
31 s3 = Student('Ben', 2)  # Student-Objekt mit name='Ben', semester=2, ID=3
32
33 # Hier wird die Methode next_semester() auf das Objekt s3 angewendet,
34 # welche den inneren Zustand des Objekts (dessen Semesterzahl) erhöht.
35 s3.next_semester()
36
37 print(s1.name)      # => Mary
38 print(s1.semester)  # => 3
39 print(s2)           # => John/5
40 print(s1 < s2)      # => True (Mary hat weniger Semester als John)
41 print(s3)           # => Ben/3 (seine Semesterzahl wurde in Zeile 35 erhöht)
42 print(Student.n)    # => 4 (die ID-Nummer für den nächsten Studenten)
```

Statt mit `__lt__` den `'<'`-Operator zu überschreiben, kann man auch eine eigene Methode definieren:

```
def less_than(self, other):
    return self.semester < other.semester
```

Nur ist dann der entsprechende Methodenaufruf `s1.less_than(s2)` nicht so gut lesbar wie `s1 < s2`.

Aufgabe 2

Beachte die Synonyme

- Objekt = Instanz
- Variable = Attribut = Eigenschaft
- Funktion = Methode

```
class Whatever:

    k = 7 # Vor Klassenvariablen muss bei Gebrauch der Klassename stehen

    def __init__(self, x): # Konstruktor mit 2 Argumenten
        self.x = x         # Argument x -> Objektvariable self.x
        self.y = 42        # Konstante 42 -> Objektvariable self.y

    def __str__(self):  # Objektmethode überschreibt die str(...)-Funktion
        return 'x={0}, y={1}'.format(self.x, self.y)  # => x=..., y=...

    def __add__(self, other): # Objektmethode überschreibt den Operator "+"
        # Erzeugt neues Whatever-Objekt mit Summe der x-Attribute
        # von self & other. (self.y ist wieder konstant 42)
        return Whatever(self.x + other.x)

    def do_this(self, a): # Objektmethode, die den Zustand des Objekts ändert
        self.x *= a # x-Attribut wird mit Faktor a multipliziert

    def do_that(b):  # Klassenmethode (kein self!) mit einem Parameter $b$
        Whatever.k += b  # b wird zur Klassenvariable addiert => neuer Wert!

w1 = Whatever(4) # => Whatever-Objekt w1 mit w1.x=4 und w1.y=42
w2 = Whatever(5) # => Whatever-Objekt w2 mit w2.x=5 und w2.y=42
w3 = w1 + w2     # => Whatever-Objekt w3 mit w3.x=9 und w3.y=42
w1.do_this(10)   # => w1 hat jetzt neu w1.x=40 und w1.y=42 (unverändert)
Whatever.do_that(3) # Klassenvariable Whatever.k hat jetzt den Wert 10

print(w1) # => 'x=40, y=42'
print(w2) # => 'x=5, 'y=42'
print(w3) # => 'x=9, y=42'
print(Whatever.k)  # => '10'
```

Aufgabe 3

Zum Lösen dieser Aufgabe genügt eigentlich nur das nötige Wissen über Vektorgeometrie, wenn man der korrekten Implementierung der dort definierten Operationen vertraut. Formal nicht korrekt sind die Ausgaben der Vektoren in den Zeilen 36–39 aber die Darstellung in Spalten- statt in Zeilenform würde hier zu viel Platz brauchen.

`v.add(w)` lässt vermuten, dass die Summe der Vektoren `v` und `w` zurückgeben wird.

`v.sub(w)` lässt vermuten, dass die Differenz der Vektoren `v` und `w` zurückgeben wird.

`v.smul(a)` lässt vermuten, dass das Produkt des Vektors `v` mit der Zahl `a` zurückgeben wird.

`v.dot(w)` lässt vermuten, dass das Skalarprodukt (eine Zahl!) der Vektoren `v` und `w` zurückgeben wird.

`v.length()` lässt vermuten, dass die Länge (=Betrag) des Vektors `v` zurückgeben wird.

```
1  class Vector:
2
3      # Konstruktor
4      def __init__(self, x, y, z):
5          self.x = x  # Weist den Komponenten die
6          self.y = y  # entsprechenden Werte von
7          self.z = z  # x, y und z zu
8
9      # Spezialmethode: # überschreiben der str-Methode
10     def __str__(self):
11         return '({0.x}|{0.y}|{0.z})'.format(self) # (x|y|z)
12
13     # Gibt Summe von self und other als Vektor-Objekt zurück
14     def add(self, other):
15         return Vector(self.x+other.x, self.y+other.y, self.z+other.z)
16
17     # Gibt Differenz von self und other als Vektor-Objekt zurück
18     def sub(self, other):
19         return Vector(self.x-other.x, self.y-other.y, self.z-other.z)
20
21     # Gibt a-faches von self als Vektor-Objekt zurück
22     def smul(self, a):
23         return Vector(a*self.x, a*self.y, a*self.z)
24
25     # Gibt das Skalprodukt von self und other zurück (ist eine Zahl!)
26     def dot(self, other):
27         return self.x*other.x + self.y*other.y + self.z*other.z
28
29     # Gibt die Länge des Vektors self zurück (ist eine Zahl!)
30     def length(self):
31         return (self.x**2+self.y**2+self.z**2)**0.5
32
33
34 a = Vector(2,2,1)
35 b = Vector(3,4,-5)
36 print(a)           # => (2|2|1)
37 print(a.add(b))    # => (5|6|-4)
38 print(a.sub(b))    # => (-1|-2|6)
39 print(a.smul(10))  # => (20|20|10)
40 print(a.dot(b))    # => 9 [Skalarpodukt: x1*x2 + y1*y2 + z1*z2 = 6+8-5]
41 print(a.length())  # => 3.0 [Länge: wurzel(x**2 + y**2 + z**2)]
```

Aufgabe 4

```
class A:

    def __init__(self, a): # Konstruktor von Klasse A
        self.a = a

    def __str__(self):
        return 'a={0}'.format(self.a)

    def f(self, x):
        return self.a + x

class B(A): # Klasse B erbt von Klasse A

    def g(self, x):
        return self.a * x

class C(B): # Klasse C erbt von Klasse B und damit auch von Klasse A

    def h(self, x):
        return self.a - x


u = A(3) # erzeugt Objekt u mit Eigenschaft u.a = 3
v = B(5) # erzeugt Objekt v mit Eigenschaft v.a = 5
w = C(7) # erzeugt Objekt w mit Eigenschaft w.a = 7

print(u) # => a=3
print(v) # => a=5
print(w) # => a=7

# Beachte: Sobald eine Methode oder ein Attribut (=Variable)
# in einer Klasse fehlt, geht Python in der Vererbungshierarchie
# eine Stufe nach oben und sucht dort nach der Ressource. Fehlt
# sie dort auch, wird weiter oben gesucht, bis sie gefunden wird.
# Erst wenn sie in der "obersten" Klasse nicht gefunden wird,
# erfolgt eine Fehlermeldung.

print(u.f(4)) # => 7 ruft Methode f [3+4] aus Klasse A auf
print(v.f(4)) # => 9 ruft Methode f [5+4] aus Klasse A auf
print(v.g(4)) # => 20 ruft Methode g [5*4] aus Klasse B auf
print(w.f(4)) # => 11 ruft Methode f [7+4] aus Klasse A auf
print(w.g(4)) # => 28 ruft Methode g [7*4] aus Klasse B auf
print(w.h(4)) # => 3 ruft Methode h [7-4] aus Klasse C auf
```

Aufgabe 5

Die Klasse `Figur` hat den Zweck, den unten definierten speziellen Figuren eine gemeinsame Identität zu verschaffen und der Klasse `Kreis` den Wert der Kreiszahl zur Verfügung zu stellen. Ferner wird hier der Kleiner-Als-Operator überschrieben, damit alle Figuren (also auch unterschidlichen Typs) aufgrund ihres Flächeninhalts verglichen werden können.

```
class Figur:

    pi = 3.14 # Klassenattribut

    def __init__(self, a, b, r): # Mehrfachzuweisung wegen Platz
        self.a, self.b, self.r = a, b, r

    def __lt__(self, other): # überschreibt den Operator "<"
        return self.inhalt() < other.inhalt()

class Rechteck(Figur):

    def __init__(self, a, b):
        super().__init__(a, b, None) # nur Attribute a und b nötig

    def __str__(self): # überschreibt str()-Funktion
        return 'a={0}, b={1}'.format(self.a, self.b)

    def inhalt(self):
        return self.a * self.b

class Quadrat(Rechteck): # erbt von 'Recheck' und so auch von 'Figur'

    def __init__(self, a):
        super().__init__(a, a) # verwendet Rechteck-Konstruktor mit b=a

    def __str__(self): # überschreibt str()-Funktion
        return 'a={0}'.format(self.a)

class Kreis(Figur):

    def __init__(self, r):
        super().__init__(None, None, r)

    def __str__(self): # überschreibt str()-Funktion
        return 'r={0}'.format(self.r)

    def inhalt(self): # erhalten pi via Vererbung aus Klasse 'Figur'
        return Kreis.pi * self.r**2

r = Rechteck(4, 5)  # => r.a=4, r.b=5, r.r=None
q = Quadrat(3)      # => q.a=4, q.b=4, q.r=None
k = Kreis(10)       # => k.a=None, k.b=None k.r=10
print(r)            # => a=4, r=5
print(q)            # => a=3
print(k)            # => r=10
print(r < q)        # => False, da r.inhalt()=20 und q.inhalt()=16
print(q < k)        # => True, da q.inhalt()=16 und k.inhalt()=314
```