Programmierworkshop

Python

Teil 2

Übungsaufgaben

und Musterlösungen

Übungen zu Funktionen

1) Schreibe eine Funktion, die aus beliebig vielen numerischen Argumenten die größte Zahl zurückgibt.

Beispiel:

```
>>> print get_max(14, 2, -1, 3, 4)
14
>>> print get_max(5, 4, 2)
5
```

2) Schreibe eine Funktion, die ihre Keyword-Argumente als Histogramm anzeigt.

Beispiel:

```
>>> histogram(europe=12, north_america=9, africa=13)
europe:        ************
north_america: *********
africa:        *************
```

Musterlösungen:

1) Übung zu positional argument lists:

```
def get_max(*nums):
    mx = nums[0]
    for m in nums:
        if m > mx:
            mx = m
    return mx
```

Alternativ auch ganz einfach die built-in Funktion max() verwenden:

```
def get_max(*nums):
    return max(nums)
```

2) Übung zu keyword argument lists:

```
def histogram(**kwargs):
    length = 0
    for k in kwargs.keys():
        if len(k) > length:
            length = len(k)

    for (k,v) in kwargs.items():
        print k + ":" + " " * (length-len(k)+1) + "*"*v
```

Kürzer gehts auch per List comprehension:

```
def histogram(**kwargs):
    length = max([len(k) for k in kwargs.keys()])
    for (k,v) in kwargs.items():
        print k + ":" + " " * (length-len(k)+1) + "*"*v
```

Übung zu Dekoratoren

Schreibe einen einfachen Dekorator um Funktionen als *deprecated* kennzeichnen zu können. Beim Aufruf einer deprecated Funktion soll eine entsprechende Warnung ausgegeben werden. Die Funktion soll aber weiterhin korrekt ausgeführt werden.

Achtung: Der Dekorator soll auf beliebige Funktionen angewendet werden können. Wie muss die Argumentliste behandelt werden?

Beispiel:

```
@deprecated
def some_old_func(x,y):
    return x+y

> print some_old_func(4,5)
Warning: Call to deprecated function some_old_func!
9
```

Musterlösung:

```
def deprecated(func):
    def wrap(*args, **kwargs):
        print "Warning: Call to deprecated function " \
              + func.__name__ + "!"
        return func(*args, **kwargs)
    return wrap


@deprecated
def some_old_func(x,y):
    return x+y

print some_old_func(4,5)
```

Übungen zur objektorientierten Programmierung

1) Grundlegende OOP in Python

Schreibe eine Klasse Katze.

Jede Katze hat einen *Namen* und ein *Alter*.

Implementiere neben dem Konstruktur eine Klassenmethode, die eine Beschreibung der Katze zurückgibt:

```
> print lizzie.get_description()
Die Katze Lizzie ist 7 Jahre alt
```

Außerdem soll eine Katzeninstanz auch miauen können:

```
> lizzie.action()
Miau!
```

Verwende New-Style Classes.

Erzeuge einige Katzeninstanzen.

2) Vererbung

a) Schreibe eine Klasse *Animal*, von der die Klasse *Katze* (und andere Tierarten) erben können. Schreibe dabei die Methode *get_description* passend als Extendermethode um. Definiere verschiedene Tierarten die von *Animal* erben.

b) Erzeuge eine Liste, die als Elemente verschiedene Tierobjekte enthält. Schreibe eine Funktion, die einen Zoo als Tabelle ausgibt. (Tipp: Den Klassennamen einer Instanz kriegt man mit:

```
type(<instance>).__name__
```

Beispiel:

```
> lizzie = Katze("Lizzie", 7)
> tiger = Katze("Tiger, 6)
> snoopy = Hund("Snoopy",8)
> jumbo = Elephant("Jumbo", 14)

> tiergarten = [lizzie, tiger, snoopy, jumbo]
> zoo_table(tiergarten)

Der Zoo enthält die Tiere:

Lizzie       7 Jahre  Katze
Tiger        6 Jahre  Katze
Snoopy       8 Jahre  Hund
Jumbo        14 Jahre Elefant
```

3) Operatorenüberladung

a) Überlade die Special Method __str__, so daß die Beschreibung der Tiere direkt mit print ausgegeben werden kann.

b) Überlade die Special Method __add__, daß die 'Addition' von zwei Tieren der gleichen Art ein Objekt der gleichen Tierart zurückgibt was 0 Jahre alt ist und den Namen trägt Name1Name2.

Falls versucht wird, zwei unterschiedliche Tierarten zu kreuzen, wird None zurückgeliefert.

Beispiel:

```
> print lizzie
Die Katze Lizzie ist 7 Jahre alt
> kaetzchen = lizzie + tiger
> print kaetzchen
Die Katze LizzieTiger ist 0 Jahre alt

> bello = Hund("Bello",5)
> hybrid = lizzie + bello
> print hybrid
None
```

4) Properties

Erweitere die Klasse Hund um ein Property *dog_age*, was beim Zugriff das Alter in "Hundejahren" angibt (Alter in Jahren * 7).

Es soll außerdem ein Docstring für dieses Property festgelegt werden.

Beispiel:

```
> bello = Hund("Bello",5)
> print bello.dog_age
35
> print Hund.dog_age.__doc__
'Alter in Hundejahren'
```

5) Statische Attribute und Klassenmethoden

Erweitere die Klasse *Animal* mit einem Attribut *num_animals* und einer Klassenmethode, so daß die Zahl von neu erzeugten Tieren jeder Tierart mitgezählt wird.

Beispiel:

```
> lizzie = Katze("Lizzie", 6)
> tiger = Katze("Tiger", 7)
> kaetzchen = lizzie + tiger
> bello = Hund("Bello",5)
> snoopy = Hund("Snoopy", 8)
> print "Es gibt jetzt", Katze.num_animals, "Katzen"
Es gibt jetzt 3 Katzen
> print "Es gibt jetzt", Hund.num_animals", "Hunde"
Es gibt jetzt 2 Hunde
```

Musterlösungen zu den Aufgaben zur objektorientierten Programmierung

Änderungen von bereits bestehendem Code vorheriger Musterlösungen ist mit **roter Textfarbe** gekennzeichnet

1) Grundlegende OOP in Python

```
class Katze(object):
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def get_description(self):
        return "Die Katze " + self.name + " ist " + \
                  str(self.age) + " Jahre alt"
    def action(self):
        print "Miau!"

lizzie = Katze("Lizzie",7)
tiger = Katze("Tiger",6)
```

2) Vererbung

a) Definition einer Elternklasse *Animal*

```
class Animal(object):
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def get_description(self):
        return self.name + " ist " + str(self.age) + \
               " Jahre alt"
```

```
class Katze(Animal):
    def get_description(self):
        return "Die Katze " + \
               super(Katze, self).get_description()
    def action(self):
        print "Miau!"

class Hund(Animal):
    def get_description(self):
        return "Der Hund " + \
               super(Hund, self).get_description()
    def action(self):
        print "Wuff!"

class Elephant(Animal):
    def get_description(self):
        return "Der Elephant " + \
               super(Elephant, self).get_description()
    def action(self):
        print "Taröööö!"
```

b) Ausgabe der Attribute einer Liste von Instanzen, mit Zugriff auf die Namen der Klasse:

```
def zoo_table(zoo):
    print "Der Zoo enthält die Tiere:"
    print
    for animal in zoo:
        print "%-10s %-2d Jahre %-10s" % \
      (animal.name, animal.age, type(animal).__name__)
```

3) Operatorenüberladung

a) Überladen von __str__:

```
class Animal(object):
    def __str__(self):
        return self.name + " ist " + str(self.age) + \
               " Jahre alt"

class Katze(Animal):
    def __str__(self):
        return "Die Katze " + \
               super(Katze, self).__str__()

class Hund(Animal):
    def __str__(self):
        return "Der Hund " + \
               super(Hund, self).__str__()

class Elephant(Animal):
    def __str__(self):
        return "Der Elephant " + \
               super(Elephant, self).__str__()
```

b) Überladen von __add__

```
class Animal(object):
    def __add__(self, other):
        if type(self) == type(other):
            return type(self)(self.name + other.name,0)
        else: return None
```

4) Properties

```
class Hund(Animal):

    def get_dog_age(self):
        return self.age * 7

    dog_age = property(get_dog_age,
              doc="Alter in Hundejahren")
```

Alternative: Kennzeichnung einer Propertymethode per Dekorator:

```
class Hund(Animal):

    @property
    def dog_age(self):
        "Alter in Hundejahren"
        return self.age * 7
```

5) Statische Attribute und Klassenmethoden

```
class Animal(object):
    num_animals = 0

    @classmethod
    def add_animal(cls):
        cls.num_animals += 1

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