lektion4

October 30, 2024

1 Modularisierung

Module (modules) sind Sammlungen von Pythonprogrammen mit der Enddung .py, die Datentypen und Funktionen bereitstellen. Module werden wieder zu Bibliotheken (libraries) oder Pakete (packages) zusammengefasst.

1.1 Einbinden von Modulen

Beispielsweise ist **numpy.random** ein Modul, das Programme enthält, die Zufallszahlen erzeugen.

[ ]:
```
from numpy.random import randint
```

[ ]:
```

```

Das math-Modul stellt mathematische Funktionen und Konstanten zur Verfügung, wie z.B. die Konstante π sowie die Funktionen sin() und cos().

Nach dem Schlüsselwort **import** können mehrere (durch Komma getrennte) Modulnamen folgen.

Obwohl import-Anweisungen an jeder Stelle des Quellcodes stehen dürfen, ist es üblich diese Anweisungen am Anfang des Quellcodes zu plazieren.

[ ]:
```
import math # importiert das math-Modul aus der Standardbibliothek (C Standard␣
↪Bibliothek)
```

[ ]:
```

```

[ ]:
```
from math import pi
```

[ ]:
```

```

Man kann auch mehrere Funktionen gleichzeitig importieren

[ ]:
```
from math import sin, cos
```

Man kann eine Bibliothek auch komplett in den globalen Namensraum einbinden. Dabei werden bereits vorhandene gleichlautende Namen überschrieben

[ ]:
```
from math import *
```

Beim Importieren einer Bibliothek kann für diese einen Alias wählen.

[ ]:
```
import math as m
import sympy as s

m.sin(m.pi), s.sin(s.pi)
```

1.2 Inhalte von eingebundenen Modulen anzeigen

Mit der built-in Funktion **dir()** kann man sich die in einem Modul definierten Namen anzeigen lassen.

[ ]:
```
import math as m
```

Mit der built-in Funktion **help()**, mit ? (jupyter) oder Ctrl I (spyder) kann man sich die Hilfe ansehen.

[ ]:
```
?m.atan2
```

[ ]:
```
help(m.atan2)
```

2 Einführung in Klassen

Eine ausführlichere Beschreibung finden Sie hier: https://www.python-kurs.eu/klassen.php

Python ist eine klassenbasierte Programmiersprache. Alle Datentypen und Funktionen sind Objekte/Instanzen von Python Klassen. Zu welcher Klasse ein Objekt gehört, erfährt man mit der **type()** Funktion sehen kann. (Die Begriffe Objekt und Instanz werden synonym verwendet.)

Selbst **None** ist ein Objekt der Klasse NoneType.

[ ]:
```

```

Eine Klasse definiert einen Bauplan nach dem die Objekte der Klasse erzeugt werden. Eine Objekt vom Typ Liste erhält man durch:

[ ]:
```
liste = list((3, 4, 5, 6))
```

[ ]:
```
print(type(liste))
```

liste ist ein Objekt der Klasse list, genauer eine Referenz auf ein Objekt der Klasse list. Allen Objekten der Klasse list stehen Attribute und Methoden zur Verfügung, die durch die Klasse definiert werden. Man kann sich diese Attribute undMethoden anzeigen lassen. Dazu schreibt man

`liste.`

plaziert den Cursor hinter dem Punkt und drückt die Tab Taste.

Es erscheint dann eine Liste möglicher Attribute und Methoden.

[ ]:
```
liste.
```

Eine vollständige Aufstellung der Attribute und Methoden kann man sich mit **dir** ausgeben lassen. Alles was mit einem Unterstrich beginnt ist privat, und wird meist nur intern verwendet.

[ ]:

Die Methode **append** kennen Sie schon.

[ ]:

Für Listenobjekte ist durch ___**add**___ eine Addition definiert. Diese funktioniert wie folgt.

[ ]:
```
liste.__add__([1, 3, 4, 1])
```

[ ]:
```
oder einfacher
```

[ ]:
```
liste + [1, 3, 4, 1]
```

Für die lineare Algebra sind Listen also nicht so einfach als Vektoren zu gebrauchen.

2.1 Eine erste minimale Klasse.

Die minimale Definition einer Klasse in Python hat die Form

```
class KlassenName:
    Anweisungen
```

[ ]:
```
class ErsteKlasse:
    """ Eine Klasse, die nichts tut """

    # pass gibt an, dass hier noch irgendwas hin soll
    pass
```

[ ]:
```
p = ErsteKlasse()
p
```

[ ]:
```
class ZweiteKlasse:
    """ eine zweite Klasse """
    # Attribut
    n = 1234

    # Methode
    def f(self, x):
        return x**2


z = ZweiteKlasse()
#z
dir(z)
```

[ ]:
```
?z
```

[ ]:
```
z.n
```

[ ]:
```
z.f(2)
```

2.2 Klasse Rechteck

[ ]:
```
class Rechteck:

    def __init__(self, L, B):
        self.laenge = L
        self.breite = B

    def __repr__(self):
        return f'{self.__class__.__name__}({self.laenge}, {self.breite})'

    def __str__(self):
        return f' {self.__class__.__name__} : {self.laenge} x {self.breite}'

    # Klassenmethoden
    def flaeche(self):
        A = self.laenge * self.breite
        return A


R = Rechteck(2, 3)
print(R)  # print nutzt die __str__ Methode
R  # hier wird die in __repr__ definierte Darstellung ausgegeben
```

Rules regarding self

1) Any class method must have self as first argument. (The name can be any valid variable name, but the name self is a widely established convention in Python.)
2) self represents an (arbitrary) instance of the class.
3) To access any class attribute inside class methods, we must prefix with self, as in self.name, where name is the name of the attribute.
4) self is dropped as argument in calls to class methods.

2.3 Vererbung

Um eine neue abgeleitete Klasse zu definieren, die von einer Basisklasse deren Methoden und Attribute erbt verwendete man

```
class DerivedClassName(BaseClassName):
    Anweisungen
```

Ein Quadrat ist ein spezielles Rechteck. Mit **super()** nutzen wir hier das ___ **init()** ___ aus der Klasse **Rechteck**

[ ]:
```
class Quadrat(Rechteck):

    def __init__(self, L):
        super().__init__(L, L)
```

Die Methoden flaeche(), ___ **repr** ___ und ___ **str** ___ erbt Quadrat von Recheteck

[ ]:
```
Q = Quadrat(2)
print(Q)
Q.flaeche()
```

[ ]:
```

```

2.4 Klasse für Polynome

[ ]:
```

```

[ ]:
```

```