Einige weitere Schritte in Python

Der Primzahlenfinder

```
limit = input("Upper limit:")
limit = int(limit)

for candidate in range(2, limit+1):
    is_prime = True
    for test in range(2, min(candidate, limit // 2+1)):
        if candidate % test == 0:
            is_prime = False
            break

    if is_prime:
        print(candidate, "is a prime")
    else:
        print(candidate, "is not a prime")
```

Etappenziel: Ordentlicherer Primzahlenfinder

- Trennen der Ein-/Ausgabe von der eigentlichen Berechnung
- Abtrennen der Berechnung ob eine einzelne Zahl eine Primzahl ist
- Alle gefundenen Primzahlen in einer Liste speichern
- Tests einbauen
- Kommandozeilenargumente

Dazu brauchen wir:

- Listen (und andere Container)
- Funktionen
- Fehlerbehandlung (Exceptions)
- Module

Listen

- Speichern mehrere Elemente
- Können nachträglich wachsen oder schrumpfen
- Werden mit eckigen Klammern gekennzeichnet
- Man kann über sie iterieren

```
einkaufen = ["milch", "eier", "kaese", "tomaten"]
for ding in einkaufen:
    print(ding, "nicht vergessen")
einkaufen.append("Aepfel")
print(einkaufen)
```

Listen: Auslesen

```
einkaufen = ["milch", "eier", "kaese", "tomaten",
             "schokolade"]
print(einkaufen)       # gesamte Liste
print(einkaufen[0])    # das erste Element (milch)
print(einkaufen[-1])   # das letzte Element
print(einkaufen[0:2]) # 1. und 2. element.
                       # Oberes Limit wie immer nicht dabei
print(einkaufen[1:])   # ab dem 2. Element
print(einkaufen[:2])   # 1. und 2. Element
                       # (oberes Limit nicht dabei)
```

Bereiche (Ranges)

- Das erste Element hat Index 0
- Von-bis wird mit von:bis geschrieben. Das obere Limit ist nicht einbegriffen
- Man kann untere und/oder obere Grenzen auslassen.
- : heißt: alles
- Vom Ende rückwärts zählt man mit negativen Zahlen
- -1 ist das letzte Element, -2 das vorletzte

Listen: Ändern

```
list.append(elem) # ein Element anhängen
list.remove(elem) # ein Element entfernen
del list[idx]     # das Element am Index idx entfernen
                  # (wieder von 0 an)
```

ACHTUNG! Copy by Reference

- In Python werden alle komplexen Datentypen als Referenz kopiert, nicht als Wert
- Ändert man die Kopie, ändert sich auch das Original und umgekehrt
- Unabhängige Kopien kann man mit `l2 = liste.copy()` erstellen
- Ausnahmen: Einfache Datentypen wie int und float

```
l1 = ["a",1.0,17]
l2 = l1
l2.append("abc")
print(l1) # ergibt ['a', 1.0, 17, 'abc']
```

aber

```
a=5
b=a
b=b+1
print(a) # immernoch 5. coy by value
```

Einschub: Hilfe bekommen und Methoden auflisten

- Für viele Funktionen und Klassen in Python gibt es eingebaute Hilfe

  ```
  help(list)
  ```

- Man kann die Methoden eines Objekts mit auflisten

  ```
  dir(list)
  ```

  - Methoden, die mit einem _ beginnen gelten als intern. Nutzung auf eigene Gefahr.
  - Methoden die mit ___ beginnen implementieren Sonderfunktionen, wie etwa die Bedeutung der eckigen Klammern und der Vergleichsoperatoren

Tupel

- In etwa wie eine Liste, aber unveränderlich
- Wird mit runden Klammern gekennzeichnet

```
a = ("x", 1)
print(x)
```

- Der Zugriff funktioniert wie bei Listen
- Auch über die Elemente eins Tupels kann iteriert werden
- Ein Tupel mit einem Element schreibt man mit Komma am Schluß, also (wert,)

Dictionaries

- Speichern Paare aus einem Schlüssel und einem zugeordneten Wert
- Ein leeres Dictionary wird mit geschweiften Klammern {} oder mit dict() erzeugt.
- Der Wert zu einem Schlüssel wird mit eckigen Klammern abgefragt

```
d1 = dict() # leer
d2 = {} # leer
# Funktionsaufruf mit Keyword-Argumenten:
d3 = dict(Montag="Monday", Dienstag="Tuesday")
# mit geschweiften Klammern:
d4 = {"Montag":"Monday", "Dienstag":"Tuesday"}
d4["Montag"] # ergibt Monday
d4["Mittwoch"] # wirft eine Exception (KeyError).
d4["Donnerstag"] = "Thursday" # hinzufügen
```

Dictionaries II

- Über ein Dictionary kann man iterieren:

```
d = dict(Montag:"Monday",Dienstag="Tuesday")
for k in d: ... # iteriert über die Schlüssel
for k in d.keys(): ... # dito
for k in d.values(): ... # iteriert über die Werte
for k, v in d.items(): ... # iteriert über Schlüssel-
                           # Wert-Paare
```

- Ist ein Schlüssel vorhanden / nicht vorhanden?

```
"Montag" in d # True
"Montag" not in d # False
"Mittwoch" in d # False
```

Funktionen

- Fassen Code in wiederverwendbarer Form zusammen
- Haben (meistens) einen Namen
- Können Eingabewerte entgegennehmen
- Und Ausgabewerte zurückgeben
- Kleine abgeschlossene Teile kann man besser testen

```
def subtract(a, b):
    return a - b

def test_subtract():
    assert subtract(1, 2) == -1
    assert subtract(5.5, 2) == 3.5

test_subtract()
print("10 - 2 =", subtract(10, 2))
```

Funktionen: Argumente

- Funktionen können zwei Arten von Argumenten haben:
  - Positional
  - Keyword

```
def point_in_circle(p, center=None, radius=None):
    return (p[0] - center[0])**2 +(p[1] - center[1])**2 \
           <= radius**2
```

- `p` ist ein positional argument, `center` und `radius` sind keyword arguments
- Keyword-Arguments stellen sicher, dass man sich bei der Nutzung der Funktion nicht bei der Reihenfolge der Argumente irrt

Funktionen: Argumente II

- Das letzte/die letzten positional argumentse können einen Default-Wert haben
- Keyword arguments haben immer eine Defaultwert. `None` wird üblicherweise verwendet, wenn es keinen sinnvollen gibt.

```
def subtract(a, b, c=0):
    return a - b - c
subtract(10, 2)
subtract(10, 2, 1)

def point_in_circle(p, center=(0, 0), radius=1.0):
    return (p[0] - center[0])**2 +(p[1] - center[1])**2 \
           <= radius**2
```

Funktionen: Argumente III

- Es ist möglich, die Anzahl der Argumente offen zu lassen

```
def f(*args, **kwargs):
    for a in args:
        print("Pos arg", a)
    for k, v in kwargs.items():
       print("keyword arg", k, "is",v)
f(1,2,first_name="John", last_name="Doe")
```

Funktionen ohne Namen und als Variablen

- Funktionen können auch wie folgt definiert werden:

  ```
  lambda x: x**2
  ```

  Funktionen können Variablen zugewiesen werden:

  ```
  subtract = lambda x, y: x - y
  subtract(10,2)
  ```

Anwendungsbeispiel: Dictionary nach Wert sortiert ausgeben

```
d = dict(a=2, b=1, c=4)
sorted(d, key=lambda x: d[x])
```

Module

- Enthalten Code den man in seinem Python-Script nutzen kann
- Man kann das Modul als ganzes, oder einzelne Funktionen daraus importieren
- Einige nützliche Module
  - sys: Systemfunktionen, etwa Kommandozeilenargumente
  - gzip: Lesen und Schreieben von .gz-komprimierten Dateien
  - argparse: Verarbeitung von Kommandozeilenargumenten
  - matplotlib: Plotten
  - numpy: Rechnen mit mehr-dimensionalen Daten
  - scipy: Wissenschaftliches Rechnen, fitten, Fourier Transformation, ...

Module importieren

- als ganzes in einen eigenen Namensraum

```
import sys
print("Kommandozeile", sys.argv)
```

- Als ganzes in einem Namensraum mit beliebigem Namen

```
import numpy as np
print(np.arange(1))
```

- Einzelne Namen aus eine Modul in den aktuellen Namensraum importieren

```
from sys import argv, executable
print("Kommandozeile", argv)
```

- Legales Python aber miese Programmierpraxis:

```
from sys import *
# jetzt weiß man nicht, welche Funktionen im aktuellen
```

- Inhalt eines Moduls anzeigen

```
import sys
dir(sys)
```

Fehlerbehandlung

- Tritt zur Laufzeit ein Fehler auf, wird eine Exception geworfen
- Wird sie nicht abgefangen, bricht das Skript mit einem Traceback ab.
- Verschiedene Fehler erzeugen unterschiedliche Exceptions
  - ValueError: Ein übergebener Wert ist nicht gültig: “a” nach int konvertieren
  - KeyError: Das Dictionary hat keinen Wert um gegebenen Schlüssel
  - IOError: Ein-/Ausgabefehler
  - ZeroDivisionError: 1/0

Fehler abfangen

- Wenn das Skript die Fehler behandeln soll, verwendet man try-except-Blöcke:

```
try:
    int("a")
except ValueError as e:
    print("Error", e)
```

- Man kann den Typ der zu fangenden Exception auslassen. Dan werden alle gefangen
- Wenn man eine Fehler still ignorieren möchte, schreibt man `pass` in den except-Block
- Man kann Exceptions auch selbst auslösen

```
raise ValueError("illegal value passed")
```

Ordentlicher Primzahlenfinder

```
def is_prime(candidate):
    is_prime = True
    for test in range(2, candidate):
        if candidate % test == 0:
            return False
    return True

def test_is_prime():
    assert is_prime(2) == True
    assert is_prime(4) == False
    assert is_prime(20) == False

def get_primes_below(limit):
    result = []
    for candidate in range(2,limit):
        if is_prime(candidate):
            result.append(candidate)
    return result
```