

Einführung in die wissenschaftliche Programmierung mit Python

Markus REINERT¹

Universität Rostock, Institut für Physik

7. Juni 2022

¹ ✉ markus.reinert@io-warnemuende.de

TIOBE Index May 2022:

Indikator für die Beliebtheit von Programmiersprachen

<https://www.tiobe.com/tiobe-index/>

- 1 **Python**
- 2 C
- 3 Java
- :
- 13 R
- 20 Matlab
- 25 Julia
- 30 Fortran



“Python plays a key role in our production pipeline. Without it a project the size of Star Wars: Episode II would have been very difficult to pull off.”

<https://www.python.org/about/quotes/>

→ **Python ist beliebt, praktisch und bestens für die Wissenschaft geeignet!**

Interaktiv im Terminal:

```

markus@meiraum:~$ python
Python 3.7.9 (default, Aug 31 2020, 12:42:55)
[GCC 7.3.0] on linux
Type "help()", "copyright()", "credits()" or "license()" for more information.
>>> print("Hello world!")
Hello world!
>>> # Python as a calculator
>>> 7 * 6
42
>>> # More complex tasks
>>> for i in range(5):
...     print("i =", i, "and i*i =", i*i)
...
i = 0 and i*i = 0
i = 1 and i*i = 1
i = 2 and i*i = 4
i = 3 and i*i = 9
i = 4 and i*i = 16
>>> institute = "IOW"
>>> institute
'IOW'
>>> print(institute)
IOW
>>> # Note the difference!

```

Als Jupyter Notebook im Internet-Browser:

The screenshot shows a Jupyter Notebook in a web browser. The notebook title is "Analyse_79NG_bathymetry". The main content area displays the title "Bathymetry around NEG and 79NG" and a brief introduction. Below the text, there are two code cells. The first cell, labeled "1 Load the dataset", contains Python code to import libraries and load a dataset. The second cell, labeled "2 Extract the bathymetric data", contains code to extract specific data from the dataset. The output of the first cell is displayed as a table with columns for dimensions, coordinates, and data variables.

```

In [1]: import xarray as xr
import numpy as np
import matplotlib.pyplot as plt
import cmocean
from scipy.interpolate import interpnd
from scipy.optimize import fsolve

1 Load the dataset

In [2]: ds = xr.open_dataset('NEG_DBR_V1_0.grd')
ds

Out[2]:
xarray.Dataset
Dimensions: (side: 2, xysize: 6480021)
Coordinates: ()
Data variables:
  x_range (side) float64 ...
  y_range (side) float64 ...
  z_range (side) float64 ...
  spacing (side) float64 ...
  dimension (side) int32 ...
  z (xysize) float32 ...
Attributes:
  title: DMagic Data Export
  source: Created by DMagic

```

Als Skript im Texteditor:

The screenshot shows a Python script in a text editor. The script defines a class "Forcing(Param)" and implements methods for calculating bathymetry and forcing terms. The code includes imports for various libraries like Fluid2d, numpy, matplotlib, and xarray. It also includes comments and docstrings to describe the code's functionality.

```

1 # Markus Reintert, 2019-06-12
2
3 from Fluid2d import Fluid2d
4 from param import Param
5 from grid import Grid
6
7 import os
8 import numpy as np
9 import matplotlib.pyplot as plt
10 import netCDF4 as nc
11
12
13 class Forcing(Param):
14     """Forcing for a double gyre."""
15
16     def __init__(self, param, grid):
17         self.x, self.y = grid.xr, grid.yr
18
19         # Basin-scale forcing: double gyre configuration
20         self.forc = -EP['tau0'] * np.sin(2 * np.pi * y / param.Ly) * grid.msk
21
22         total = grid.domain_integration(self.forc)
23         self.forc -= (total / grid.area) * grid.msk
24
25         # Buffer zone West
26         self.buffer_scale = 1 / EP['b_tline'] + 1/2 * (1 + np.tanh(
27             (x - param.Lx + EP['b_wldth']) / EP['b_extend']
28             ))
29         self.buffer_scale *= grid.msk
30
31     def add_forcing(self, x, t, dxdt):
32         """Add the Forcing term on x[0]=the vorticity."""
33         dxdt[0] += self.forc - self.buffer_scale * (x[0] - self.pvback)
34
35
36 # Set fundamental model parameters
37 param = Param(ens_file="M4.exp")
38 for EP in param.loop_experiment_parameters():
39     # Set type of model and domain, its size and resolution
40     param.modelname = "quasigeostrophic"
41     param.geometry = "closed"
42     param.nx = 128 * EP['resolution']
43     param.ny = 128 * EP['resolution']
44     param.Lx = 2560
45     param.Ly = 2560
46     # Set number of CPU cores used
47     if os.uname()[1] == "apus":
48         print("### Working on apus, using four cores.")
49         param.npx = 2
50         param.npy = 2
51     else:
52         param.npx = 1
53         param.npy = 1
54
55 -----
56 !---- M4.py Top (1,0) (Python 11 Elpy)
Mark set

```

Operation	Operator	Beispiel
Addition	+	1 + 2
Subtraktion	-	2 - 1
Multiplikation	*	3 * 4
Division	/	9 / 4 (gibt 2.25)
Ganzzahl-Division	//	9 // 4 (gibt 2)
Modulo-Rechnung	%	9 % 4 (gibt 1)
Potenzrechnung	**	3 ** 2 (gibt 9)
Wurzelziehen	**(1/2)	9 ** (1/2) (gibt 3.0)

Beachte:

Integer (Ganzzahlen) werden automatisch in *Floats* (Fließkommazahlen) umgewandelt, wenn es notwendig ist oder wenn ein Operand ein Float ist.

... also eigentlich gibt es doch nichts zu beachten (außer man benutzt Python 2).

Außerdem: nur runde Klammern () dürfen in Rechnungen verwendet werden.

Strings definiert man wahlweise so: "Ich bin ein Text" oder so 'Ich auch'.

Und wenn der Text Anführungszeichen enthält?

1. "Er sagte 'Hallo'." (doppelte Anführungszeichen sind „unsichtbar“)
2. 'Er sagte "Hallo".' (einfache Anführungszeichen sind „unsichtbar“)
3. 'Er sagte \'Hallo\''.' (equivalent zu 1.)
4. "Er sagte \"Hallo\"." (equivalent zu 2.)

Und wenn der Text besonders lang ist?

- ▶ `"""Dieser Text geht über drei (!) Zeilen
und kann " sowie ' enthalten.
"""`
- ▶ hauptsächlich als Kommentar / Beschreibung (einer Funktion *etc.*) verwendet

Mit Strings kann man rechnen:

- ▶ `"Hallo " + "Welt"` ergibt `"Hallo Welt"`
- ▶ `3 * "wiu"` ergibt `"wiuwuiwui"`

Achtung: `"4" + "2"` ergibt `"42"` und `"1" * 5` ergibt `"11111"`

Um mathematisch zu rechnen:

- ▶ `int("4") + int("2")` ergibt 6
- ▶ `float("0.5") + float("0.5")` ergibt 1.0
- ▶ andersrum: `str(3/2)` ergibt `"1.5"`

Hilfreiche Funktionen:

- ▶ `" Wörter mit Leerzeichen ".strip()` ergibt `"Wörter mit Leerzeichen"`
- ▶ `"Das_ist_ein_Text".replace("_", " ")` ergibt `"Das ist ein Text"`

Gute Möglichkeiten um Zahlen in Text einzufügen:

1. `"The density is {:.1f} kg/m3".format(1030.49)`
 - ▶ die Zahl im Argument von `format` wird verwendet
- 2a. `f"The density is {rho:.1f} kg/m3."`
 - ▶ der Wert der Variable `rho` wird verwendet
- 2b. `f"The density anomaly is {rho-1000:.1f} kg/m3."`
 - ▶ einfache Rechnungen sind möglich
 - ▶ komplexe Rechnungen auch, aber dafür ist evtl. Möglichkeit 1 übersichtlicher

Erklärungen:

- ▶ die Zahl wird anstelle der geschweiften Klammern eingefügt (`f` für Float)
- ▶ die Zahl wird auf 1 Nachkommastelle gerundet (wegen `.1`)
- ▶ und es gibt noch viele weitere Formatierungsmöglichkeiten

Variablen:

- ▶ werden in Python normalerweise mit = definiert
- ▶ können beliebige Daten/Objekte enthalten:
 - ▶ Zahlen: `x = 3`
 - ▶ Strings: `text = "Hallo"`
 - ▶ Listen, Funktionen, ...
- ▶ können Namen aus Buchstaben, Zahlen und Unterstrichen haben, aber Namen dürfen nicht mit einer Zahl beginnen
- ▶ sollten **sinnvolle** Namen haben

Konstanten:

- ▶ gibt es in Python nicht
- ▶ jede Variable kann überschrieben bzw. verändert werden
- ▶ Konvention: Variablennamen in GROSSBUCHSTABEN stehen für Konstanten

Liste

Sammlung beliebiger Daten

```
temp = [20, 22, 20.5, "error", 21]
```

meist für *unbekannte* Anzahl von Daten
desselben Typs verwendet

Tuple

unveränderbare Liste

```
dimensionen = (128, 256, 32)
```

meist für *bekannte* Anzahl von Daten
unterschiedlichen Typs verwendet

Named Tuple – oft besser geeignet als ein einfaches Tuple:

```
# zu Beginn des Programms:
```

```
from collections import namedtuple
```

```
# bevor das Tuple verwendet wird:
```

```
Dimension = namedtuple("Dimension", ("x", "y", "z"))
```

```
# Definition des NamedTuples:
```

```
dim = Dimension(128, 256, 32)      # oder: Dimension(x=128, y=256, z=32)
```

```
# Verwendung:
```

```
dim.x      # ergibt 128
```

```
dim.z      # ergibt 32
```

- ▶ Anzahl der Elemente in einer Liste: `len(daten)`
- ▶ `daten[i]` gibt Element `i` aus Liste `daten` zurück
(geht auch mit `NamedTuples`, sollte dort aber vermieden werden, siehe vorherige Folie)
- ▶ `daten[i] = x` überschreibt Element `i` der Liste `daten` mit dem Wert von `x`
(geht nicht mit `Tuples` oder `NamedTuples`)
- ▶ **Wichtig:** Zählung beginnt bei `i = 0`
Bsp.: hat eine Liste 10 Elemente, so können Indizes 0 bis 9 verwendet werden

- ▶ `personal_data = {"name": "Max", "age": 23, "city": "Rostock"}`
- ▶ Zugriff auf ein Element: `personal_data["name"]`
- ▶ Zugriff auf alle Elemente: `personal_data.items()`

For-Schleife:

feste Anzahl an Wiederholungen, z. B.:

- ▶ n -mal
- ▶ für jedes Element einer Datensammlung
- ▶ für jede Datei in einem Ordner

While-Schleife:

Anzahl der Wiederholungen geknüpft an eine Bedingung, z. B.:

- ▶ Konvergenz eines numerischen Verfahrens
- ▶ sequentielles Lesen von Daten

Wichtig:

Jede Anweisung, die zu einer Schleife gehört, muss mit derselben Anzahl an Leerzeichen eingerückt sein!

Üblich sind 4 Leerzeichen oder ein Tab.

Ausnahme: einzelne Anweisungen können in derselben Zeile stehen.

Vergleichsoperatoren:

- ▶ Gleichheit: ==
- ▶ Ungleichheit: !=
- ▶ Vergleich: <, >, <=, >=
- ▶ Verknüpfung: and, or, not

Mehrere Vergleiche:

```
if x > 0:  
    print("x ist positiv")  
elif x == 0:  
    print("x ist null")  
else:  
    print("x ist negativ")
```

Wichtig: richtige Einrückung beachten (wie bei Schleifen).

- ▶ können manchmal praktisch sein
- ▶ sollten Lesbarkeit verbessern
- ▶ können Funktionen ersetzen (z. B. `map`)

Beispiele:

- ▶ `half_data = [x / 2 for x in data]`
- ▶ `inverse_data = [1 / x if x != 0 else 0.0 for x in data]`
- ▶ `print(
 "The influence is",
 "significant" if p <= alpha else "not significant",
)`

Anweisungen, die an mehr als einer Stelle verwendet werden und eine bestimmte *Funktion* erfüllen, können als Funktion definiert werden.

Funktionen haben Eingabewerte (Argumente) und Ausgabewerte.

Arten von Funktionsargumenten:

Argumente ohne Namen

Beispiel:

```
print(3, "ist kleiner als", 4)
```

Optionale Namen

Beispiel:

```
np.zeros((3, 4))  
# ist equivalent zu  
np.zeros(shape=(3, 4))
```

Argumente mit Namen

Beispiel:

```
print("Dieser Text", end=" ")  
# Argument "end" kann nicht  
# ohne Namen verwendet werden  
print("geht hier weiter.")
```

- ▶ **Verwende die Schreibweise mit Namen, wenn es die Lesbarkeit verbessert!**
- ▶ Argumente ohne Namen müssen immer vor Argumenten mit Namen kommen.
- ▶ Die Reihenfolge von Argumenten ohne Namen ist von Bedeutung.
- ▶ Die Reihenfolge von Argumenten mit Namen ist egal.

Einfachstmögliche Funktion:

```
def square(x):  
    return x ** 2
```

Kompliziertere Funktion:

```
def gauss(x, mu=0.0, sigma=1.0):  
    """Calculate the value of the Gauss distribution with mean mu and std.dev. sigma at x.  
  
    x: float  
    mu: mean of the distribution (default 0)  
    sigma: standard deviation (default 1)  
    """  
    if sigma == 0:  
        print("Error!")  
        return np.NaN # beendet den Funktionsaufruf  
    z = (x-mu) / (2*sigma)  
    return np.exp(- z ** 2) / np.sqrt(2*np.pi*sigma**2)
```

Sammlung von (veränderlichen) Daten:

Listen, (Named) Tuples,
Dictionaries, andere
Datenstrukturen

Sammlung von Funktionen (und ggf. Konstanten):

Module (= Bibliotheken):
Python-Dateien,
die man mit `import` in ein
Programm einbinden kann

Sammlung von Daten mit zugehörigen Funktionen:

Klassen

“Programs are meant to be read by humans
and only incidentally for computers to execute.”
(Harold und Sussman)

- aussagekräftige Namen wählen
- Leerzeichen einbauen
- lange Zeilen umbrechen, am Besten innerhalb von Klammern
- Einrückungen verwenden
- Kommentare hinzufügen *# in Python geht das so*

Weitere Tipps:

<https://www.python.org/dev/peps/pep-0020/>

<https://google.github.io/styleguide/pyguide.html>