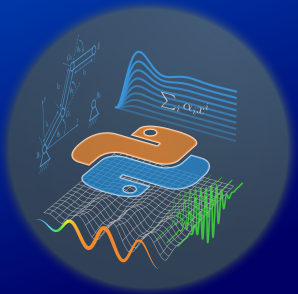
Carsten Knoll
Chair of Fundamentals of Electrical Engineering

Python for Engineers
Pythonkurs für Ingenieur:innen

Advanced Programming Techniques
Fortgeschrittene Programmierkonzepte

Dresden (Online), 2023-11-28

https://tu-dresden.de/pythonkurs
https://python-fuer-ingenieure.de



Outline

- functional programming,
- nested functions
- namespaces
- `import` mechanisms
- exceptions (error handling)
- PEP8, unittests, documentation tools, version management

Functional Programming (and List Comprehensions)

- application of functions to sequences (e.g. lists)
  - keyword `lambda` → anonymous function (“throwaway function”)

```
L = [1, 3, 5, 7, 9]
squares = list(map(lambda z: z**2, L))
big_numbers = list(filter(lambda n: n > 5, L))
# map and filter each return iterators -> convert to list
```

- compact syntax but not so easy to read/understand
- recommended alternative for `map` filter `filter`: **list comprehension**:

```
squares = [z**2 for z in L]
big_numbers = [z for z in L if z > 5]
```

- function application for matrices and arrays:
  `numpy.ufunc`, `sympy.Matrix.applyfunc`, `pandas.DataFrame.apply`
- typical in functional programming: **recursion** (≙ function that calls itself)

```
def factorial(n):  # calculate n!
    if n == 0:
        return 1
    else:
        return n*factorial(n-1)
```

Nested Functions (I)

- functions are ordinary objects

→ can be arguments of other function (like `solve_ivp(rhs, ...)` in course03).

```
def func1(x):
    "a docstring"
    return x + 1

print(type(func1)) # -> <type 'function'>

def read_the_docs(f):
    print(f.__doc__)

read_the_docs(func1) # -> "a docstring"
```

Nested Functions (II)

- functions can also be **created** within other functions.
- (important: scopes of variables (“scopes”))

```
def factory(y):
    def f(x):
        return x + y

    return f # return the function object

func1 = factory(10)
func2 = factory(5)

print(func1(2), "; ", func2(2)) # -> 12 ;  7
```

Decorators (I)

"wrapper functions": functions that are applied to functions and return new functions

Listing: example-code/01_decorator.py

```
result_cache = dict()

def cached(func):

    def wrapper(arg):
        key = (func, arg)
        if key in result_cache:
            print("use cached result")
            return result_cache[key]
        else:
            res = func(arg)
            result_cache[key] = res
            return res
    return wrapper

def func1(x):
    print("Executing func1 with x =", x)
    return x**2

wrapped_func1 = cached(func1)

# original function gets called
print(wrapped_func1(5))

# value from the cache gets used
print(wrapped_func1(5))
```

"wrapper": (german: Verpackung, Umhüllung)

Example:

- memory for results of calculations
- memory `dict` object
- key: 2-tuple: (function, argument)
- value: return value of function
- motivation: saving computation time

General:

- decorators are useful for reusable pre- and postprocessing of data

Decorators(II)

- outer (factory) function (i.e. the wrapper) is called “decorator”
- special short syntax to wrap a function: use `@`
- skip the variable assignment :
  `wrapped_func = decorator(original_func)`
- original function is not available under its own name

Listing: example-code/01_decorators.py (28-34)

```
@cached
def func2(x):
    print("Executing func2 with x =", x)
    return 100 + x

print(func2(4))  # original function gets called
print(func2(4))  # value from the cache gets used
```

- decoators can also be applied to methods and classes
- builtin decorators: `staticmethod`, `classmethod`, ...
- more information on decorators:
  - programiz.com/python-programming/decorator
  - thecodeship.com/patterns/guide-to-python-function-decorators/

Namespaces

- namespace: scope of variable names.
- each name is used at most once per namespace → uniqueness
- terms "'namespace'" and "'scope'" almost synonymous
- common: nested namespaces
  - in a normal function: 2 levels (global and local namespace)
  - function in function: 3 levels
- Python interpreter searches names from inside to outside
  if a name is not found in the local scope the next higher scope is searched
- each module has a global namespace
- keep order and overview with prefixes:

```
import math  # from Python's standard lib (no array support)
import numpy  # with array support
x = 1.23
a = numpy.arrange(4)

print(math.exp(x))
print(numpy.exp(a))
# these funcs have the same name, but need to be separated
# math.exp(a) would cause an error
```

Importing Modules and Packages

- module = Python file
- currently executed file: “main module”
- `import`: import names (+ objects bound to them) into own namespace
- two options:

```
# option 1: import the module
import numpy
x = numpy.arange(5)

# option 2: import single objects into global namespace
from numpy import arange, pi
x = arange(5)*pi
```

- abbreviations (aliases) with keyword `as`:

```
# option 1: import the module
import numpy as np
x = np.arange(5)

# option 2: import single objects into global namespace
from numpy import arange as ar
x = ar(5)
```

Imports (II)

Other abbreviations:

```
# this is possible but considered bad style
# multiple imports in one line (with or without alias):
import numpy as np, matplotlib, sympy
```

- wildcard import (*)
- Only recommended for interactive testing
  - (no control which names are imported and overwritten if necessary):

```
from numpy import *  # "namespace pollution"
```

Imports (III)

- imported module is executed "normally":
  (but only once; `importlib.reload` → force true re-import)

```
# module1.py
def function1(x):
    return x**2 - 3*x + 5

print("abc")  # will produce output upon importing this module
Z = 123
```

```
# main.py      (located in the same directory as module1.py)
import importlib
import module1  # -> abc
import module1  # no new output
importlib.reload(module1)  # -> abc
print(module1.Z + 0.4)  # -> 123.4
module1.function1(0)  # -> 5
```

- modules are also objects:

```
print(type(module1)) # -> <type 'module'>
```

Packages / Import Paths

- module = Python file
- package = directory containing file `__init__.py` and possibly other modules
- can be nested: `package.subpackage1.subpackage2.module.object`
- file `__init__.py` is allowed to be empty

What is the purpose of modularization?

- reusability, redistribution, exchange
- thematic structuring of code

Packages / Import Paths

- module = Python file
- package = directory containing file `__init__.py` and possibly other modules
- can be nested: `package.subpackage1.subpackage2.module.object`
- file `__init__.py` is allowed to be empty

What is the purpose of modularization?

- reusability, redistribution, exchange
- thematic structuring of code

Import paths:

- Python interpreter searches in certain default directories (incl. current workdir)

```
import sys
print(sys.path)  # -> ['', '/anaconda/lib/python3.8/site-packages', ...]
```

- customization options:
  - `PYTHONPATH` environment variable
  - `my_path.pth` file (in one of the default directories)
  - `sys.path.append(..)`

Exceptions (Error Handling)

- errors are inevitable during programming
- good error handling saves a lot of time and nerves
- Python: “exceptions”
  - are "'thrown'" at an erroneous position
  - can be "caught" in an error location

```
def F(x, y):
    return x/y
def G(z):
    try:
        print(F(10, z))
    except ZeroDivisionError:
        print("Division by 0 is not allowed!")
G(5) # -> 2
G(0) # -> Division by 0 is not allowed!
```

- important types: `Exception` (=base class for all exceptions), `NameError`, `ValueError`, `TypeError`, `AttributeError`, `NotImplementedError`,
- more information: https://docs.python.org/3/tutorial/errors.html

Exceptions (II)

Recommendations:

- every source code is based on assumptions made during development
- occasionally check if assumptions are still true at runtime

→ throw exceptions in your own code: `raise`

```
def F(x):
    if not isinstance(x, (float, int)):
        msg = f"number expected but got {type(x)}"
        raise TypeError(msg)
    return x**2
```

- more convenient (⇒ lower usage hurdle): `assert`

```
def F(x):
    assert isinstance(x, (float, int))
    return x**2
```

→ exception type: `AssertionError`

- disadvantage: unspecific error (type information missing) → only for “private use”.

Further Topics

Python **E**nhancement **P**roposal #8; short: PEP8, python.org/dev/peps/pep-0008/

- style guide for good code (convetions for naming and formatting)

`unittest` package, https://docs.python.org/3/library/unittest.html

- automated testing of your code
- indispensable for larger projects, already pays off for small projects

`sphinx` package, http://www.sphinx-doc.org/en/stable

- automatically generate documentation from code (and docstrings).

→ requires good docstrings

version control with Git

- essential for projects with multiple people but also useful on its own
- recommendation: https://git-scm.com, see also FSFW-git-intro-workshop (de)

Further Topics

Python **E**nhancement **P**roposal #8; short: PEP8, python.org/dev/peps/pep-0008/

- style guide for good code (convetions for naming and formatting)

`unittest` package, https://docs.python.org/3/library/unittest.html

- automated testing of your code
- indispensable for larger projects, already pays off for small projects

`sphinx` package, http://www.sphinx-doc.org/en/stable

- automatically generate documentation from code (and docstrings).

→ requires good docstrings

version control with Git

- essential for projects with multiple people but also useful on its own
- recommendation: https://git-scm.com, see also FSFW-git-intro-workshop (de)

still more topics: typing hints, continuous integration, meta class programming, ...

Summary

- functional programming (`lambda, map, filter`, list comprehension).
- nested functions (define functions within functions)
- decorators
- namespaces, `import` mechanisms (keeping order)
- exceptions (throwing and catching, `assert`)
- PEP8, unittests, doc tools, version management