HOWTO zu den Spezialitäten von Python



Dieses Dokument beschreibt die Besonderheiten von Python im Vergleich zu
anderen Programmiersprachen oder Skriptsprachen.

---

```
* Conceived as TEACHING/LEARNING/TRAINING language (in the beginning)
  --> Easy to learn syntax
  --> Indentation counts --> Makes Copy-and-Paste difficult
  --> Documentation easily integratable
  --> Educational aspects important (e.g. indentation, very clear error messages)

* FULLY object oriented programming language (OOP)
  + EVERYTHING is an OBJECT (even numbers, functions, classes, modules, ...)
    --> Functions, classes, modules, ... are "FIRST CLASS" objects!
        Can be: created at runtime
                passed as parameters to and returned from functions
                assigned to variables
  + Each built-in DATATYPE is a CLASS
    --> Self defined CLASSES behave like built-in datatypes!
    --> May be used to inherit from
  + BASE CLASS of ever class is "object"
  + All MEMBER FUNCTIONS are VIRTUAL
  + All MEMBERS are PUBLIC (no real encapsulation)
    --> Naming conventions cause PRIVATE/PROTECTED members
  + DUCK TYPING: if it looks and behaves like a duck, it's a duck
                 interface is the same --> undistinguishable
  + MONKEY PATCHING (classes/instances may be dynamically changed)

* EVERYTHING (EACH OBJECT)
  + Has a fixed DATATYPE: type(OBJ)
  + Has a fixed unique ID: id(OBJ) = memory address
  + Has a REFERENCE COUNTER (counts names pointing to it): sys.refcounter(OBJ)
  + May be converted to STRING by str(OBJ) / repr(OBJ)
  + May be converted to BOOL by bool(OBJ)
  + May be printed out by print(...)
  + Has a BOOLEAN VALUE True/False in boolean context
  + May be compared to any other object by ==     (value equal)
                                       and !=     (value different)
  + May be compared to any other object by is     (identical object)
                                       and is not (different object)
  + May have ATTRIBUTES (key-value pairs) associated with it
    (built-in datatypes NoneType int float complex str tuple list dict don't!)

* SYNTAX
  + INDENTATION is part of the syntax + defines NESTING STRUCTURE (BLOCK)
    (colon ":" <-> ONE indented statement needed --> keyword "pass" if empty)
    --> Pretty-printer (automatic indentation) impossible! --> do it yourself!
    --> No automatic indentation by IDE/Tool possible!
    --> Only ignored between parentheses ( [ { ... } ] )
                     between multiline string quotes """..."""
                     in empty lines and before comments #....
                     next line after line continuation "\" at line end
  + One line = one statement (normally)
  + No special statement terminator but line end
    (but ";" is statement separator to combine several statements on one line)

* Token = Keywords + Operators + Identifiers + ...
  + UPPER/lower case counts EVERYWHERE (identifier, keyword, module name, ...)
  + 35 KEYWORDS (only) have a fixed meaning (all other IDENTFIERS allow change)
  + 75 BUILT-IN FUNCTIONS (non-OOP, may change their meaning, but shouldn't)
  + 55 OPERATORS mapped to "magic methods" --> redefinable for own datatype
  + 94 MAGIC METHODS (called automatically by built-in function, operator,
    object creation, iteration, function entry/exit, ...)
  + Identifier
    - XXX_ used as identifier if XXX is a KEYWORD
    - __XXX__ are python INTERNAL names ("MAGIC METHODS, there are a lot of them!)
    - __XXX are private names of classes (mangled --> _CLASS__XXX)
    - _XXX are protected names of classes or not exported names of modules
    - _ used as syntactically necessary identifier if value not needed
    - _ contains result of last expression in interactive interpreter
    - _ often used with internationalization (i18n) and localization (l10n)
```

```
* EVERYTHING is an OBJECT (even numbers, functions, classes, modules, ...)
  --> Functions are "FIRST CLASS" objects!

* Each DATATYPE is a CLASS
  --> Self defined CLASSES behave like built-in datatypes!

* Each VALUE/OBJECT/INSTANCE knows it's DATATYPE + number of REFERENCES to it
  --> Automatic type checking during program run
  --> Automatic reference counting + object destroyance + garbage collection!

* IDENTIFIER are just REFERENCES to OBJECTS (SYMBOL TABLE entry)
  (means VARIABLES store references to OBJECTS)
  --> So Variables are ALWAYS initialized
  --> So any identifier may point to any object during run-time!
  --> Any identifier may be redefined any time!
  --> Any identifier may be deleted by "del ..." (removed from symbol table)!

* DATATYPE of VALUE is defined by VALUE SYNTAX or explicit DATATYPE CONVERSION
  --> No variable declaration (but TYPE HINTS since Python 3.5/3.6/3.7)

* NO AUTOMATIC DATATYPE CONVERSION --> has to be done MANUALLY --- but:
  + Numeric Types int <-> float <-> complex <-> bool in expressions
    (boolean True/False --> 1/0 in expressions)
  + ANY DATATYPE may be converted --> bool (e.g. in boolean context if ...:)
  + ANY DATATYPE may be converted --> str (e.g. autom. in function print())

* EACH OBJECT
  + Has a datatype: type(OBJ)
  + Has a unique id: id(OBJ) = memory address
  + Has a reference counter: contains  number of references to it
  + May be converted to a STRING by str(OBJ) / repr(OBJ)
  + May be printed out by print(...)
  + Has a boolean value True/False in boolean context
  + May be compared to any other object by == (value eqal)
                                        and != (value different)
  + May be compared to any other object by is (identical object)
                                        and is not (different object)
  + May have ATTRIBUTES (key-value pairs)

* Lots of RUN-TIME CHECKS (automatically and permanent)
  + Access/usage of values datatype + functions + operators
  + Access/usage of index/key
  + Access/usage of mutable/im-mutable = read-write/read-only datatypes
    --> NoneType bool int float complex str bytes tuple frozenset ...
  + Datatype conversion possible
  + Operator applyable to operand datatypes
  + Reference counter == 0 --> Object may be destroyed and its memory freed

* Any RUN-TIME ERROR cancels program execution and prints out
  + Script filename
  + Line number
  + Error class (e.g. "FileNotFoundError")
  + Error message (e.g. "division by zero not allowed")
  + Traceback (call stack = way through function calls to error code line)

* Error handling is always done by exception handling or context object
  --> "try-except" and "with"
  --> Separate "real" code and error handling

* Datatype names may be used as FUNCTION to do CONVERSION to that datatype
  (e.g. datatype int --> conversion function int("1234") --> 1234)
  + Create Objects from Class-Name

* Impossible CONVERSIONS are not allowed
  + "None" cannot be used in expressions
  + Any data from outside is always of datatype "str" (argv, environ, ...)
  + i = int(input("Please give a number: ")) crashes on input of a float "1.0"

* Functions
  + Definition + call ALWAYS need PARENTHESES (...)
    --> WITHOUT PARENTHESES --> reference to funktion object!
  + Always have a RETURN VALUE (at least "None") which may always be ignored
  + Allow ANY OBJECT as parameter or return value
  + Allow positional and named parameters
  + Allow necessary and optional parameters
```

```
  + Allow any number of positional/named parameters
  + Decorators = wrap function by "enhancer function" (cascadable)

* Lot of SEQUENCES (indexed, ordered, similar behaviour, similar syntax)
  + str       = sequence of chars            (read-only)
  + bytes     = sequence of bytes            (read-only)
  + tuple     = sequence of elements/objects (read-only)
  + list      = sequence of elements/objects (read-write)
  + bytearray = sequence of bytes            (read-write)
  + file      = sequence of lines separated by "\n" or "\r\n")

* Tries to delay/retard any work as long as possible
  + Call by reference
  + Assignment --> COW = Copy on Write (late binding)
  + Tuple/list/dictionary comprehension
  + Iterators
  + Generators (map, filter, reduce, zip, ...)

* DON'T COUNT yourself, let python do it for you via
  + for-loop over sequences or collections or files
  + for (i,v) in enumerate(SEQ): ...
  + Function range(N,M,S)
  + Function slice(N,M,S)
  + Slicing [N:M:S]

* DOCUMENTATION very easy
  + Integrated via DOCSTRINGS into source code (reStructured)
  + Generatable from source code via "pydoc" or "easydoc" or "Sphinx"
  + Done by ASCII or reStructured or ... text

* REFLECTION / SELFINSPECTION possible
  + Function type()
  + Function id()
  + Function dir()
  + Function help()
  + Function callable()
  + Function isinstance()
  + Function issubclass()
  + List of variables in namespace by vars() globals() locals()
  + Attributes: __name__ __class__ __weak__ __call__
  + Attribute dictionary: __dict__
  + Symbol table dictionary: __dir__ (Namespace)
  + Attribute access: getattr() setattr() hasattr() delattr()
  + Class Method Resolution Order: CLASS.__mro__ CLASS.mro()

* Declarative instead of procedural programming
  + Tuple/List/Dictionary comprehension (declarative instead of functional)
  + Generators
  + Decorators

* Specialities
  + Datatypes are IM-MUTABLE/READ-ONLY (bool int float complex str tuple)
                or MUTABLE/READ-WRITABLE (list dict set)
  + Only one type of value transfer: CALL BY REFERENCE
    --> Always references are used/moved (NEVER VALUES)
  + Assignment ASSIGNS new reference to variable name (COW = copy on write)
  + Memory allocation/deallocation done by python itself (garbage collection)
  + There is no empty statement, keyword "pass" needed
  + "else" may be used at the end of most control structures
    (if, for, while, try, with, ...)
  + String technique if identifer "no yet" usable but needed
    -  __slots__
    -  getattr, setattr, delattr, hasattr, ...
```