HOWTO Specialities of Python

This document describes the specialities of Python compared to other
programming or script languages.

---

* Conceived as TEACHING/LEARNING/TRAINING language (in the beginning)
  --> Educational aspects important
  --> Easy to learn syntax (e.g. no block braces, no statement terminators)
  --> Indentation counts --> makes Copy-and-Paste difficult
  --> One line = one statement
  --> Documentation easily integratable
  --> Functions are "FIRST CLASS" objects!
      (same USAGE and BEHAVIOUR as DATA)

* FULLY object oriented programming language (OOP)
  + EVERYTHING is an OBJECT (number, string, function, datatype, class, module)
    --> Number, function, datatype, class, module 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!
    --> Usable as base class for inheritance
  + BASE CLASS of each class is "object" (nice name!)
  + All MEMBERS are PUBLIC (no real encapsulation)
    --> Real ENCAPSULATION possible by naming conventions and __slots__

* FULLY DYNAMICAL
  + All MEMBER FUNCTIONS are VIRTUAL
  + DUCK TYPING: if it looks and behaves like a duck, it's a duck
                 same interface --> undistinguishable
  + MONKEY PATCHING: classes/instances may be dynamically changed

* SYNTAX
  + UPPER/lower case counts EVERYWHERE (identifier, keyword, module name, ...)
  + INDENTATION is part of syntax + defines NESTING STRUCTURE (BLOCK)
    (colon ":" <-> indented statement(s) 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 comment lines #....
                    in lines after line with line continuation "\" at end
  + One line = one statement (normally)
  + No special statement terminator but line end
    (";" may separate statements to combine several ones on one line)

* Token = Keywords + Operators + Identifiers + ...
  + 35 KEYWORDS (only) have a fixed meaning (all other IDENTIFIERS may change)
  + 75 BUILT-IN FUNCTIONS (GENERIC, non-OOP, may change 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, ...)
  + Identifiers are classified by "NAMING CONVENTIONS" --> PEP8
    - Use XXX_ as identifier if XXX is a KEYWORD (may be no good idea)
    - __XXX__ are 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
    - XXX are PUBLIC names of classes

```
    - _ used as syntactically necessary identifier if value not needed
    - _ contains result of last expression in interactive interpreter
    - _ often used for internationalization (i18n) and localization (l10n)

* 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 destroying + garbage collection!

* IDENTIFIER contain just REFERENCES to OBJECTS (SYMBOL TABLE entry)
  (means VARIABLE stores reference to OBJECT)
  --> 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 HINT/ANNOTATION since Python 3.5-3.10)

* 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 automatically converted to bool in boolean context if/while ...:
  + ANY DATATYPE automatically converted to str by function print(...)
  + ANY DATATYPE comparable by "==" "!=" "is" "is not" to any other DATATYPE

* EACH OBJECT
  + Has a DATATYPE:                                          type(OBJ)
  + Has a UNIQUE ID (memory address):                        id(OBJ)
  + Has a REFERENCE COUNTER (counts names pointing to it): sys.getrefcount(OBJ)
  + Has a memory size (in bytes):                            sys.getsizeof(OBJ)
  + May be converted to STRING by:                           str(OBJ) repr(OBJ)
ascii(OBJ)
  + May be PRINTED out:                                      print(OBJ)
  + May be converted to BOOL:                                bool(OBJ)
  + Has a boolean value True/False in BOOLEAN CONTEXT:       if while and or not
  + May be COMPARED BY VALUE to any other object by ==       (type AND value equal)
                                                  and !=     (type OR value different)
  + May be COMPARED BY ID to any other object by is          (identical object)
                                               and is not    (different object)
  + May have ATTRIBUTES (key-value pairs) associated with it
    (not for built-in datatypes because of space and performance reasons:
     NoneType int float complex str tuple list dict set frozenset bytes bytearray ...)

* 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)
  + Catching via "try...except" necessary to continue program

* Error handling always done by exception handling or context object
  --> "try-except" and "with"
  --> Clear separation of "real" code and "error handling" code
```

```
* Datatype name usable:
  + to CREATE OBJECT of that type: class Robot --> r1 = Robot(...)
  + as CONVERSION FUNCTION to that datatype (e.g. int("123") --> 123 (int))

* Impossible CONVERSIONS are not allowed
  + "None" cannot be used in expressions
  + Data from outside is always of datatype "str" (sys.argv, os.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 function object!
  + Always have a RETURN VALUE (at least "None") which may be ignored
  + Allow ANY OBJECT as parameter or return value (symmetric)
  + Allow positional and named parameters
  + Allow necessary and optional parameters
  + Allow any number of parameters
  + Decorators = wrap function by "enhancer function" (cascadable)
  + No function OVERLOADING possible (SIGNATURE = just function name)
    (but DISPATCHING via analysing number/type of parameters)

* Lot of SEQUENCES (indexed, ordered, similar behaviour, same 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" (read or write)
  + array     = sequence of int/float numbers                  (read-write)

* 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

* 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)
  + slicing [N:M:S]

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

* REFLECTION / INTROSPECTION / SELFDESCRIPTION possible
  + Function type()
  + Function id()
  + Function dir()
  + Function help()
  + Function callable()
  + Function Attributes __code__ __defaults__ __kwdefaults__ __annotations__
__closure__
  + Function isinstance()
  + Function issubclass()
  + List of variables in namespace by globals() locals() vars()
  + Attributes: __name__ __qualname__ __class__ __weakref__
  + Attribute dictionary: __dict__
  + Attribute slots: __slots_
  + Documentation: __doc__
  + Symbol table dictionary: __dir__ (Namespace)
  + Attribute access: hasattr() getattr() setattr() delattr()
```

```
  + Iterator protocol: iter()  next()  send()  throw()  StopIteration
  + Generator protocol: yield  (comprehension)
  + Buffer protocol:
  + Descriptor protocol: __get__() __set__() __delete__()

* Declarative instead of procedural programming
  + Generator/List/Dictionary/Set Comprehension (declarative instead of functional)
  + Decorators

* Specialities
  + Datatypes are IM-MUTABLE/READ-ONLY (bool int float complex str tuple bytes
frozenset)
                or MUTABLE/READ-WRITABLE (list set dict bytearray)
  + 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 several control structures
    (if, for, while, try, with, ...)
```