About Python

Prof. Dr. Thomas Kopinski

2019

why Python

- easy to accomplish things quickly, compare ‚Hello World‘ in Python vs. Java

Python

```
>>> print ('Hello World')
Hello World
```

Java

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, world!");
    }
}
```

- lots of packages for data wrangling - numpy, pandas etc.
- easy to visualize data and results of analysis (matplotlib)
- popular industry-approved frameworks and kits to do *machine learning,* i.e. tensorflow, pybrain, scikitlearn and many more
- transfer code into C with Cython (runtime reduction)
- connect to MySQL databases easily (PyMySQL)
- BeautifulSoup for web scraping
- iPython for interactive programming , jupyter notebooks is a modern and easy to use environment

About Python

- **high-level** programming language
  - as opposed to low-level languages Python is more readable and therefore the scope of programming becomes wider
- **high readability** due to simple syntax:
  - makes creating prototypes easier
  - fast tests of ideas
  - easy readability for other programmers —> easy to write, easy to read
- **object-orientation**
  - OO languages allow for reusing of concepts
  - modern way of programming
- cross-platform
- large number of modules included
- widely supported esp. in ML / DS
- add-on modules, libraries, frameworks tool-kits etc.

Python comparison

Code Conversion

- Python interpreter converts code to machine code line by line
- Java compiler converts the source code into bytecode

Data types

- Python: data types exist implicit (dynamic), no need to declare
- C/C++, Java require variables to have a data type

Statements

- Python: statements are within a line
- C/C++, Java requires semicolons at the end of statements

Python comparison

Speed

- Python: is typically slower than other high-level languages such as C++

- Main reasons:
  - Python is interpreted at runtime (**cf**: runtime errors), others are typically compiled
  - no primitives in Python —> everything including builtin types (`float, str`) is an object
  - Python lists can hold various types —> therefore every entry in a list needs to store the datatype somewhere additionally!
  - but: scientific modules (scipy, numpy) are pretty fast!

- **CPython**:
  - most popular Python implementation is implemented in C
  - still CPython compiles high-level Python code into bytecode
  - compared to compiling low-level C directly into machine code

Python - implementation

Implementation

- find out which implementation you are using:

```
import platform
print(platform.python_implementation())
```

```
CPython
```

- **CPython**:
  - original implementation of Python
  - Guido van Rossum's reference version of the Python computing language
  - most popular Python implementation implemented in C
  - still CPython compiles high-level Python code into bytecode
  - compared to compiling low-level C directly into machine code

Python - implementation

Implementation

- **CPython**:
  - term is used to distinguish CPython (implementation of the language engine) from Python (the programming language)
  - CPython translates Python Code into Bytecode
  - it interprets this bytecode in an evaluation loop
  - others:
    - Jython: compiles Python code into *Java bytecode* so it can run on the JVM
    - Cython: Project which translates Python code to C (another page)
- **Bytecode**:
  - code processed by a program (usually VM), not the real computer machine (the hardware processor)

Python - bytecode

```python
import dis

def f(x):
    s = 'string'
    y = x + 10
    print (s)
    print (y)

dis.dis(f)
```

```
35           0 LOAD_CONST               1 ('string')
             2 STORE_FAST               1 (s)

36           4 LOAD_FAST                0 (x)
             6 LOAD_CONST               2 (10)
             8 BINARY_ADD
            10 STORE_FAST               2 (y)

37          12 LOAD_GLOBAL              0 (print)
            14 LOAD_FAST                1 (s)
            16 CALL_FUNCTION            1
            18 POP_TOP

38          20 LOAD_GLOBAL              0 (print)
            22 LOAD_FAST                2 (y)
            24 CALL_FUNCTION            1
            26 POP_TOP
            28 LOAD_CONST               0 (None)
            30 RETURN_VALUE
```

Bytecode

- CPython bytecode
- not portable
- bytecode produced in CPython is specific for the interpreter
- easy for further execution of code

Python - implementation example list

Implementation

- from the C implementation of a list

```c
#ifndef Py_LIMITED_API
typedef struct {
    PyObject_VAR_HEAD
    /* Vector of pointers to list elements.  list[0] is ob_item[0], etc. */
    PyObject **ob_item;

    /* ob_item contains space for 'allocated' elements.  The number
     * currently in use is ob_size.
     * Invariants:
     *     0 <= ob_size <= allocated
     *     len(list) == ob_size
     *     ob_item == NULL implies ob_size == allocated == 0
     * list.sort() temporarily sets allocated to -1 to detect mutations.
     *
     * Items must normally not be NULL, except during construction when
     * the list is not yet visible outside the function that builds it.
     */
    Py_ssize_t allocated;
} PyListObject;
#endif
```

- in: https://github.com/python/cpython/ —> include —> listobject.h
- list is an array of pointers

Python - implementation example list

```c
static int
list_resize(PyListObject *self, Py_ssize_t newsize)
{
    PyObject **items;
    size_t new_allocated, num_allocated_bytes;
    Py_ssize_t allocated = self->allocated;

    /* Bypass realloc() when a previous overallocation is large enough
       to accommodate the newsize.  If the newsize falls lower than half
       the allocated size, then proceed with the realloc() to shrink the list.
    */
    if (allocated >= newsize && newsize >= (allocated >> 1)) {
        assert(self->ob_item != NULL || newsize == 0);
        Py_SIZE(self) = newsize;
        return 0;
    }
```

- code for resizing of an array if it is full

```c
    new_allocated = (size_t)newsize + (newsize >> 3) + (newsize < 9 ? 3 : 6);
    if (new_allocated > (size_t)PY_SSIZE_T_MAX / sizeof(PyObject *)) {
        PyErr_NoMemory();
        return -1;
    }

    if (newsize == 0)
        new_allocated = 0;
    num_allocated_bytes = new_allocated * sizeof(PyObject *);
    items = (PyObject **)PyMem_Realloc(self->ob_item, num_allocated_bytes);
    if (items == NULL) {
        PyErr_NoMemory();
        return -1;
    }
    self->ob_item = items;
    Py_SIZE(self) = newsize;
    self->allocated = new_allocated;
    return 0;
}
```

- resizing of array with new size

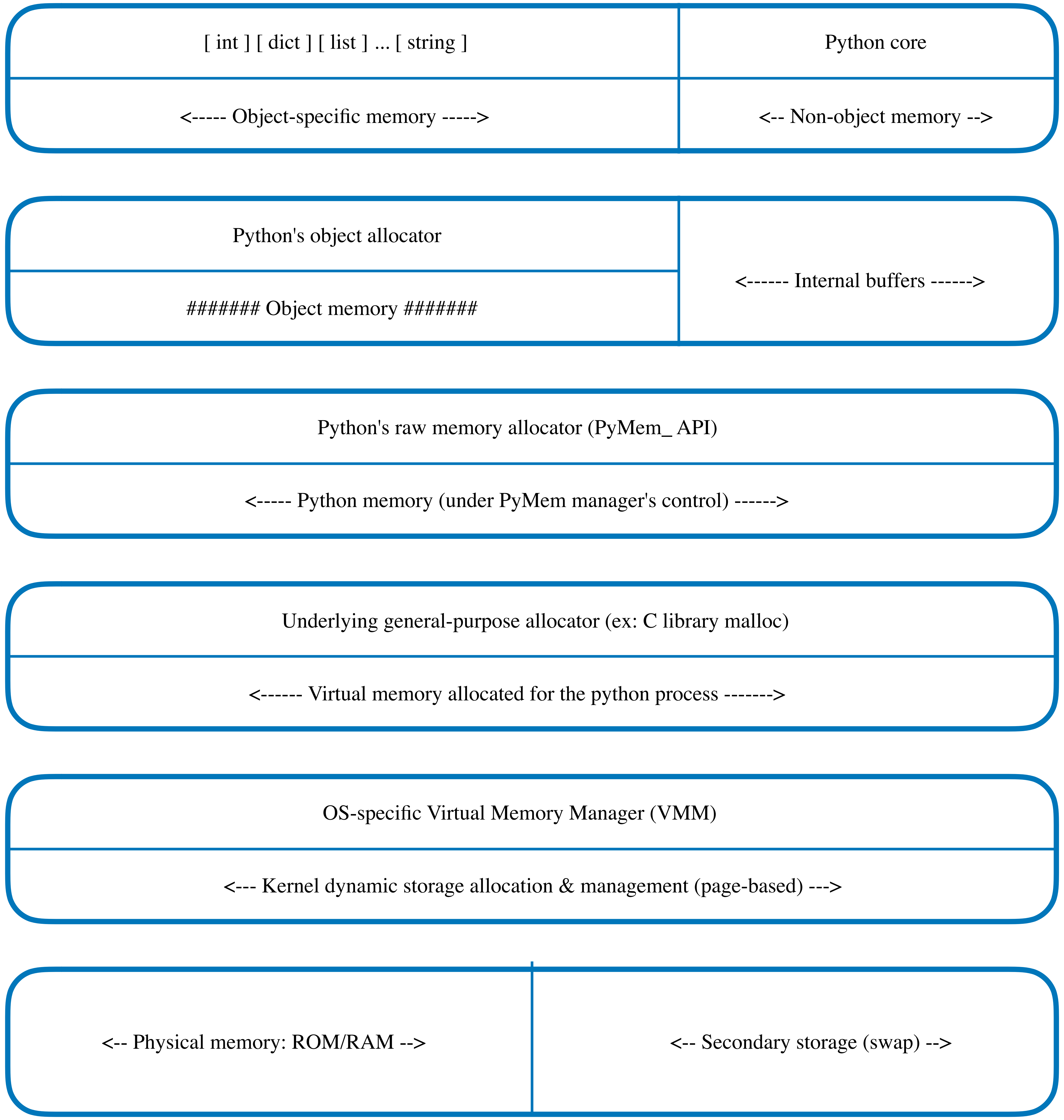
Python - memory management

Memory

- <u>Python</u> is typically weaker compared to the consumption of memory of other languages

- Main reasons:
  - data types are flexible —> increased memory overhead
- everything is an object in Python
- objects can hold other objects (lists, tuples etc.)
- —> many small memory allocations required
- hence: PyMalloc — special manager on top of the general-purpose allocator
- also good to reduce fragmentation

Python - memory management



[https://rushter.com/blog/python-memory-managment/]

Python - memory management

- large objects are routed to the standard C allocator
- for small objects (<512bytes) : sub-allocation of large blocks of memory
- three levels of abstraction:
  - *block*
  - *arena*
  - *pool*

Block

Request in bytes	Block allocation size	Size class idx
1-8	8	0
9-16	16	1
17-24	24	2
25-32	32	3
33-40	…	…
…	…	…
505-512	512	63

- Block: Chunk of memory of fixed size, can only keep one Python object
- size of a block can vary depending on the size, size is a multiple of 8

Python - memory management

Pool

- collection of blocks of the <u>same size</u>
- avoiding of fragmentation, if we force the pool to be consisting of fixed block sizes —> if an object gets destroyed, we fill it with a new object of the same size

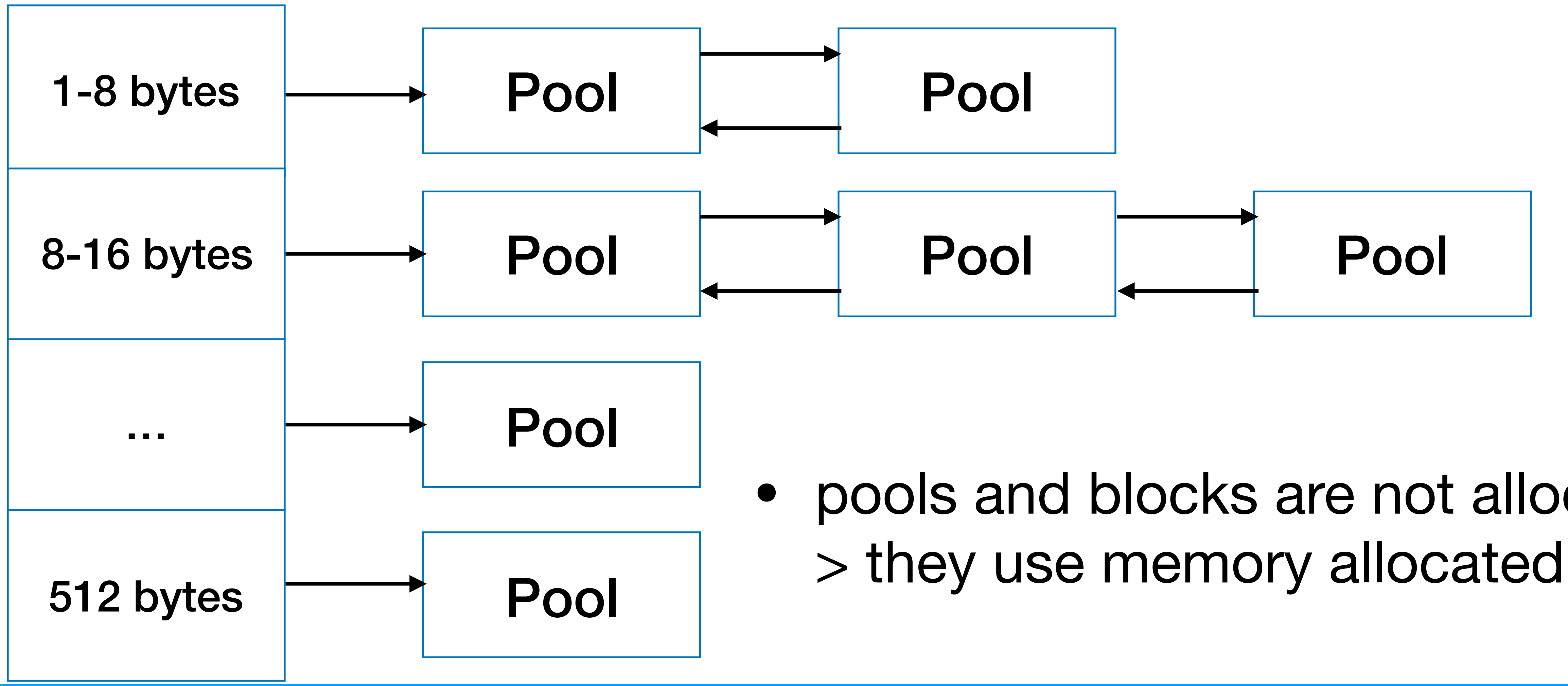
```c
/* Pool for small blocks. */
struct pool_header {
    union { block *_padding;
            uint count; } ref;          /* number of allocated blocks    */
    block *freeblock;                   /* pool's free list head         */
    struct pool_header *nextpool;       /* next pool of this size class  */
    struct pool_header *prevpool;       /* previous pool       ""        */
    uint arenaindex;                    /* index into arenas of base adr */
    uint szidx;                         /* block size class index        */
    uint nextoffset;                    /* bytes to virgin block         */
    uint maxnextoffset;                 /* largest valid nextoffset      */
};
```

- linked list: nexpool and prevpool links pools of the same size
- `szidx`: keeps block size
- `arenaindex`: stores the number of the arena where the pool was created
- `freeblock`: if a block is empty, instead of an object, the address of the next empty block is stored —> saves memory and computation

Python - memory management

Pool

- each pool has three states:
  - used - partially used (not full, not empty)
  - full - all blocks allocated
  - empty - all blocks in the pool are available for allocation
  - additional array: `usedpools`
    - stores pointers to the pools grouped by class
    - pools of same block size are already linked
    - for iteration over the pools we only need start of the list
    - if no such pool is present, then one is created of the requested size



- pools and blocks are not allocating memory themselves —> they use memory allocated from arenas

Python - memory management

Arena

- arena: chunk of 256kB memory allocated on the heap
- provides memory for 64 pools

pool: 4kb	pool: 4kb	pool: 4kb	pool: 4kb
pool: 4kb	pool: 4kb	pool: 4kb	pool: 4kb
pool: 4kb	free pool: 4kb	free pool: 4kb	free pool: 4kb
free pool: 4kb	free pool: 4kb	...	...

Python - memory management

Arena

```c
struct arena_object {
    /* The address of the arena, as returned by malloc.  Note that 0
     * will never be returned by a successful malloc, and is used
     * here to mark an arena_object that doesn't correspond to an
     * allocated arena.
     */
    uintptr_t address;

    /* Pool-aligned pointer to the next pool to be carved off. */
    block* pool_address;

    /* The number of available pools in the arena:  free pools + never-
     * allocated pools.
     */
    uint nfreepools;

    /* The total number of pools in the arena, whether or not available. */
    uint ntotalpools;

    /* Singly-linked list of available pools. */
    struct pool_header* freepools;

    struct arena_object* nextarena;
    struct arena_object* prevarena;
};
```

- arenas are linked with doubly linked lists
- `ntotalpools` and `nfreepools` - information about currently available pools
- freepools field points to the linked list of available pools
- i.e.: list of containers which automatically allocates new memory for pools when needed

Python - memory management

Memory deallocation

- small object manager: rarely returns memory to the OS
- arena is released, iff all the pools in it are empty
- example: use of temporary objects in a short time
- therefore: long running Python processes may hold a lot of unused memory bc. of the behaviour above

Python - memory management

Mobile

- Python is not very well suited for mobile applications
- not the main focus

IoT / Embedded

- development here is better done with C/C++

Runtime errors

- these errors occur frequently due to dynamical typing
- requires more testing

no ++/- -

Python - advantages

readability

- sytax easy to understand, read, write, learn
- complex concepts are more advanced, not as easy
  (list comprehensions, extra packages etc.)
- dynamical typing - can lead to errors, problems with scoping

Python - advantages

duck typing

- duck test: *If it looks like a duck, swims like a duck, and quacks like a duck, then it probably is a duck.*
- functionality of a class is more important than its type

```
class MyClass:
    def __len__(self):
        return(2000)

print(len('this'))
print(len([1,2,3]))
print(len(MyClass()))
print(len(5.0))
```

- e.g.: any class which defines a `.__len()__` method can be used to call `len()` on
- this works independent of the object's type

```
4
3
2000
Traceback (most recent call last):
  File "examples.py", line 8, in <module>
    print(len(5.0))
TypeError: object of type 'float' has no len()
```

Python - advantages

indentation

- blocks are defined by indentation
- language requirement, not a matter of style
- statements with the same indentation belong to the same block, i.e. align vertically

- nested indentations: new blocks within blocks are indented further
- do not mix tabs and spaces - suggestion: use whitespace
  —> text editors on different systems behave diffferently and problems follow
- 4 whitespaces == 1 indent
- indentations cannot be split (e.g. by backslashes)
- PEP -8: https://www.python.org/dev/peps/pep-0008/#tabs-or-spaces

1. indent

2. indent

3. indent

```python
def func(str):
    l = []
    for c in str:
        if c not in ['a', 'e', 'i', 'o', 'u']:
            l.append(c)
    s = set(l)
    return s

print(func('find a set from this string'))
```

```
{'m', 'h', 'f', 'n', 'r', ' ', 'd', 's', 't', 'g'}
```

Python - advantages

FLOSS

- Free / Libre and Open Source Software
- freely available, download, code and go!

High-level

- no need for memory management
- no need to deal / remember the system architecture

Portability

- develop on a Unix-machine, run the code on a Windows machine

Python - advantages

Interpretation

- code is executed line by line
- internally it is first converted to bytecode

Other

- large number of libraries (good for dara science, machine learning)
- GUI easy to make
- embeddable into other languages like C++

Python scripts

General

- no need to retype everything into the shell
- store Python code in .py files
  - start the interpreter
- alternatively —> use notebooks
- use any text editor to create python code
- compiled CPython files are bytecode files

Specifics

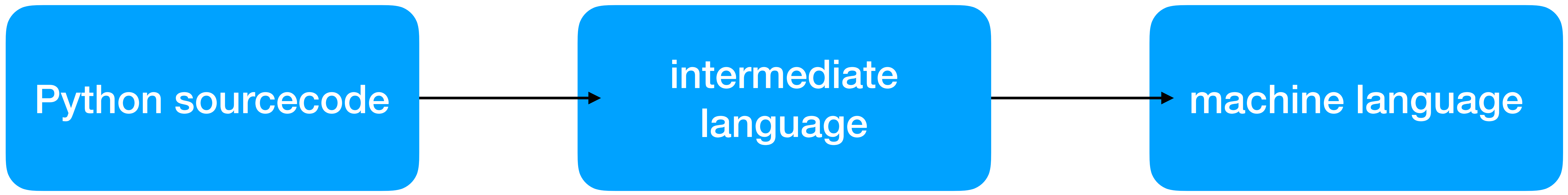
- comments follow the *hashtag* #
- variables are namespaces (`var1, myVar`) to store some values of a data type

Python interpreter

- Python files are assumed to be in `ASCII`
- other encodings are to be assigned at the beginning of a script by e.g.:
  - `# -*- utf-8 -**`
- on unix machines the default Python interpreter can be found under
  - `/usr/local/bin`
- on Windows e.g.:
  - `C:\Python3.3`

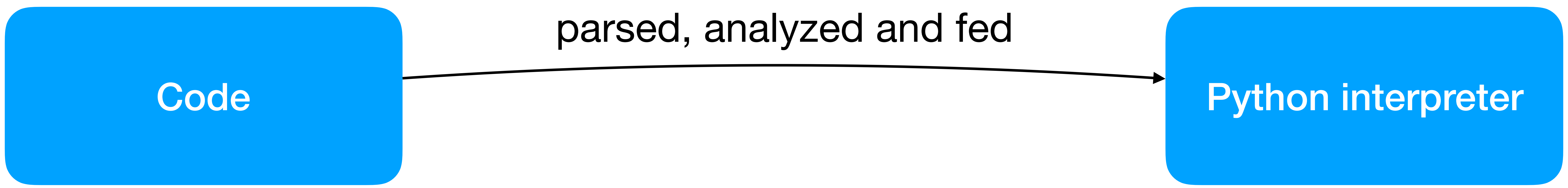
Python interpreter

- interpreter is a program
- executes other programs



- Python code —> Python bytecode (`.pyc`)
- lower-lever, platform-independent representation
- (`.pyc`) files are executed by virtual machines which carry out each operation
- VM is the runtime engine of Python
- technically: VM is the last step of the **Python interpreter**

Python interpreter

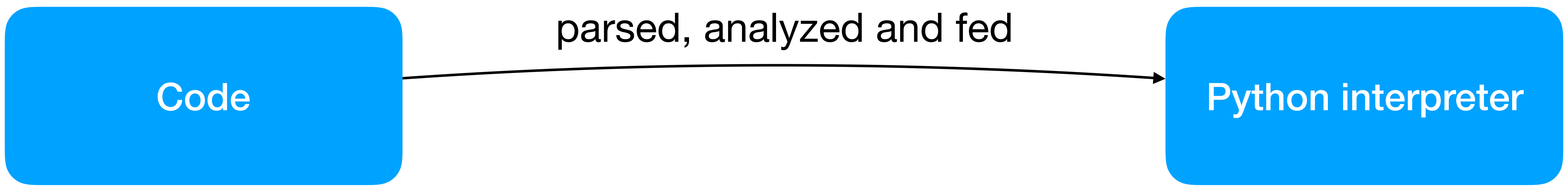


- a compiler starts with a parser:
  - apply syntax rules of the language
  - result make no sense? —> error, abort
  - result ok? —> produce an Abstract Syntax Tree (AST)
  - program is represented in nodes + edges of an AST
- next step: semantic analysis
  - check for unallowed usage
  - e.g.: wrong function calls on objects
  - if no error - edit the tree for machine usage
- final step: a valid, simplified AST is fed into the generator
  - produce code in output language
- Python: uses an **<u>interpreter</u>**
  - same as a compiler
  - one difference: instead of code generation it loads the output in-memory and executes it directly on the system

Python interpreter

- Python runtime system/library
  - contains a large set of important functions used frequently
  - set of functions collected in a shared lib
  - code can call into this lib at runtime —> *runtime library / the runtime*

Python interpreter



- a note from the glossary:
  - **bytecode**: Python source code is compiled into bytecode, the internal representation of a Python program in the CPython interpreter.
    The bytecode is also cached in `.pyc` and `.pyo` files so that executing the same file is faster the second time (recompilation from source to bytecode can be avoided). This “intermediate language” is said to run on a virtual machine that executes the machine code corresponding to each bytecode. Do note that bytecodes are not expected to work between different Python virtual machines, nor to be stable between Python releases.