SYSTEMS DEVELOPMENT FOR COMPUTATIONAL SCIENCE

LECTURE 20

Fabian Wermelinger
Harvard University
CS107 / AC207

Tuesday, November 8th 2022

LAST TIME

- Generators
- Coroutines

TODAY

Main topics: ***Python internals and the interpreter loop***

Details:

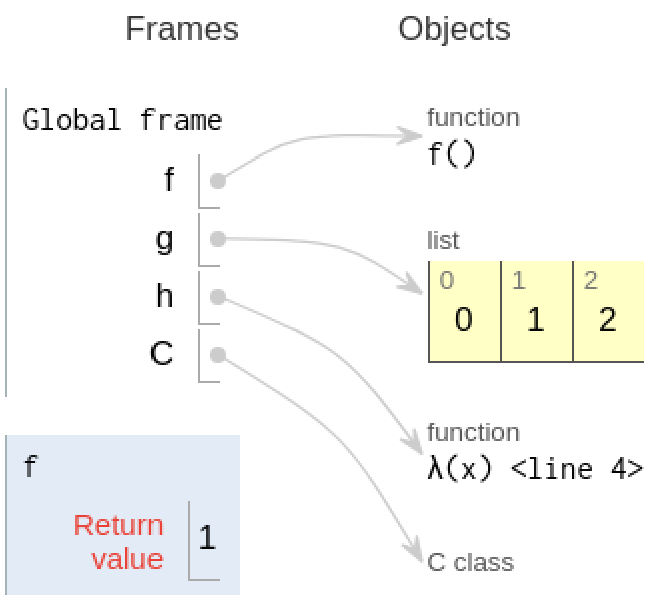
- Python internals:
  - Code objects and bytecode
  - The interpreter and the evaluation loop
  - Frame objects
  - Generator objects
- Why are Python built-in lists slow and NumPy arrays fast?

AGENDA CHECK:

- Quiz 3 takes place on Thursday. ***You will have 25 minutes*** for 12 questions. The quiz is available within a 12 hour time window (starting 9:00am). Question topics: https://edstem.org/us/courses/24296/discussion/2039027
- Milestone 2B due on Thursday. The milestone consists of a brief progress report https://harvard-iacs.github.io/2022-CS107/project/M2B/. Please limit to no more than 1/2 page.

PYTHON INTERNALS: OBJECTS AND FRAMES

Recall: the sketches we saw during the pythontutor examples



- ***Frames:*** "frame objects" that execute code (imagine a *stack* data structure: the blue shaded frame is at the top of the stack).
- ***Objects:*** any other Python objects. Functions, classes, data structures, etc.

- Frame objects *execute* code and form a *sequence* in a *stack* data structure.
- Arrows indicate *references to objects in memory*.
- When we enter a function `f()`, a new *frame* is *pushed* onto the stack and executes (blue shaded frame on left).
- When done, the function frame is *popped* off the stack and we return to the caller frame (global frame on left).
- The data structure used to organize frames is a LIFO stack. ***Will that work for coroutines?***

PYTHON INTERNALS: OBJECTS

All the data stored in a Python program is built around the concept of an object.

Terminology:

- Every piece of data is stored in an *object*. *This includes Python frames and code.*
- Each object has an *identity*, a *type* (also known as its class) and a *value*.
- The identity of an object is its location in memory. *Names* store a *reference* to a specific memory location.
- The *type* of an object describes the internal representation as well as methods and operations it supports → implemented in a `class`.
- When an object of a specific type is created, we called it an *instance* of that type. *After an instance is created, its identity and type can no longer be changed.*
- If an object's value can be modified, we call it *mutable*, otherwise it is said to be *immutable*.
- *Containers* or *collections* are objects that contain references to other objects.
- Because everything in Python is represented by objects, they are said to be *first class*.

PYTHON INTERNALS: OBJECTS

Example: user defined function object

- User defined functions are *callable* objects created at the module level by using `def` or `lambda`. Functions are ***first class*** objects in Python.

```
>>> def f():
...     pass
...
>>> g = lambda x: x
>>> f.__code__; g.__code__
<code object f at 0x7fd88a3fac90, file "<stdin>", line 1>
<code object <lambda> at 0x7fd88a3fabe0, file "<stdin>", line 1>
```

- A user-defined function `f` has the following attributes:

Attribute	Description
`f.__doc__`	Documentation string
`f.__name__`	Function name
`f.__dict__`	Dictionary containing function attributes
`f.__code__`	Byte-compiled code
`f.__defaults__`	Tuple containing the default arguments
`f.__globals__`	Dictionary defining the global namespace
`f.__closure__`	Tuple containing data related to nested scopes

- Python code ***is compiled*** into *bytecode* objects *on the fly*.
- Python is an *interpreted* language → under the hood, however, code is transformed into *bytecode objects*. The interpreter is a virtual machine.
- Running your code for the first time is slower due to bytecode generation. The result is *cached* in `.pyc` files for faster subsequent execution.

PYTHON INTERNALS: CODE OBJECTS

- *The Python interpreter executes code objects. They represent raw bytecode.*
- We can generate code objects with the `compile()` built-in function:

```
>>> a = 1
>>> co = compile('a + 1', '<string>', mode='eval')
>>> eval(co) # evaluate the code object
2
```

- The *raw bytecode* is contained in `co_code`:

```
>>> co.co_code
b'e\x00d\x00\x17\x00S\x00'   # raw binary encoded Python bytecode
```

- We can *disassemble* bytecode into the *instructions* that Python executes for a particular code object:

```
>>> import dis
>>> dis.dis(co)
  1           0 LOAD_NAME                0 (a)
              2 LOAD_CONST               0 (1)
              4 BINARY_ADD
              6 RETURN_VALUE
```

4 instructions are executed: *2 loads*, *1 binary addition* and *returning* the result. A list of all bytecode instructions can be found here.

PYTHON INTERNALS: INTERPRETER

- All the data stored in a Python program is built around the concept of an *object*. Code objects are *compiled bytecode*. The interpreter turns those code objects into *frame objects* and executes them (*left column in pythontutor*).
- Bytecode is an implementation detail of the CPython interpreter and ***not portable*** between different Python versions → *ignore `__pycache__` and `.pyc` files in your Git repositories!*
- → to execute frame objects, *input* data is required (*right column in pythontutor*).
- The CPython interpreter obtains input data from a *value stack* and executes frame objects arranged in a *frame stack* in a central loop called the ***evaluation loop***. In the *interactive* Python shell this is called ***REPL***: **R**ead, **E**valuate, **P**rint, **L**oop. The CPython interpreter is written in C → https://github.com/python/cpython
- At the ***very core*** of the evaluation loop is the `_PyEval_EvalFrameDefault` function. This is the function that brings everything together and makes your code come to life. *Everything that is executed in Python must go through this function*. Recent addition of specialized instructions in Python 3.11 have improved this function. See PEP 659 for more details.

PYTHON INTERNALS: NAME SCOPES

- We saw that a `LOAD_NAME` instruction pushes the object at given index in `co_names` onto the *value stack*. This instruction obtains the object from the *local scope*. The `LOAD_GLOBAL` instruction would be used to load a name from the *global scope*.
- In Python, the *local* and *global* scopes can be inspected with the `locals()` and `globals()` built-ins, respectively:

```
>>> def f(x):
...     l0 = x  # l0 and x are local names
...     l1 = g  # g is a global name
...     print(f'id(x) = {id(x)}')
...     print(f'id(g) = {id(g)}')
...     print(f'Local vars in f(x):  {locals()}')
...     print(f'Global vars in f(x): {globals()}')
...
>>> g = 42
>>> f(g)
id(x) = 140268417435152  # local and global names hold same reference!
id(g) = 140268417435152  # local and global names hold same reference!
Local vars in f(x):  {'x': 42, 'l0': 42, 'l1': 42}
Global vars in f(x): {'__name__': '__main__', '__doc__': None, other special attributes,
'__builtins__': <module 'builtins' (built-in)>, 'f': <function f at 0x7f92c937fd90>, 'g': 42}
```

PYTHON INTERNALS: NONLOCAL SCOPE

- Recall the `nonlocal` statement of Problem 4 in Homework 2.
- Python code objects have special attributes for names in scopes:
  - `co_varnames`: names of ***local variables***.
  - `co_cellvars`: names of local variables that are ***referenced by nested functions***.
- The `nonlocal` statement in Python is implemented based on these attributes and the corresponding *instructions*:

```python
def global(x): # not same `x`!
    def c(y):  # closure
        global x # not same `x`!
        x -= y # x = x - y
        return x
    return c
```

```python
def nlocal(x): # same `x`!
    def c(y):  # closure
        nonlocal x # same `x`!
        x -= y # x = x - y
        return x
    return closure
```

```python
def local(x):
    def c(y):  # closure
        x -= y # x = x - y
        return x
    return c
```

```
Cell vars `global`:  ()
Local vars `global`: ('x', 'c')
Local vars `c`:      ('y',)
  4     0 LOAD_GLOBAL        0 (x)
        2 LOAD_FAST          0 (y)
        4 INPLACE_SUBTRACT
        6 STORE_GLOBAL       0 (x)

  5     8 LOAD_GLOBAL        0 (x)
       10 RETURN_VALUE
```

```
Cell vars `nlocal`:  ('x',)
Local vars `nlocal`: ('x', 'c')
Local vars `c`:      ('y',)
  4     0 LOAD_DEREF         0 (x)
        2 LOAD_FAST          0 (y)
        4 INPLACE_SUBTRACT
        6 STORE_DEREF        0 (x)

  5     8 LOAD_DEREF         0 (x)
       10 RETURN_VALUE
```

```
Cell vars `local`:  ()
Local vars `local`: ('x', 'c')
Local vars `c`:     ('y', 'x')
  3     0 LOAD_FAST          1 (x)
        2 LOAD_FAST          0 (y)
        4 INPLACE_SUBTRACT
        6 STORE_FAST         1 (x)

  4     8 LOAD_FAST          1 (x)
       10 RETURN_VALUE
```

PYTHON INTERNALS: NONLOCAL SCOPE

What is causing the UnboundLocalError?

```python
def local(x):
    def c(y):  # closure
        x = x - y # same as x -= y
        return x
    return c
```

```
Cell vars `local`:  ()
Local vars `local`: ('x', 'c')
Local vars `c`:     ('y', 'x')
 3    0 LOAD_FAST            1 (x)
      2 LOAD_FAST            0 (y)
      4 INPLACE_SUBTRACT
      6 STORE_FAST           1 (x)

 4    8 LOAD_FAST            1 (x)
     10 RETURN_VALUE
```

- Python assumes `x` is contained in the local scope because it "*sees*" an assignment to `x`.
- Python can use more efficient instructions this way → `LOAD_FAST`
- This instruction will fail however because `x` ***is not bound to the local scope!***
- Python raises a `UnboundLocalError` because it cannot load `x` at bytecode offset `0`!
- The `nonlocal` statement will put `x` into the `co_cellvars` tuple and use the `LOAD_DEREF` instruction instead.

PYTHON INTERNALS: NONLOCAL SCOPE

Without assignment → only read x:

```
def local(x):
    def c(y):  # closure
        y = x  # read-only x!
        return y
    return c
```

```
Cell vars `local`:  ('x',)
Local vars `local`: ('x', 'c')
Local vars `c`:     ('y',)
 3    0 LOAD_DEREF         0 (x)
      2 STORE_FAST         0 (y)

 4    4 LOAD_FAST          0 (y)
      6 RETURN_VALUE
```

- The name `x` is non-local but can be referenced from the outer scope.
- Python realizes this automatically and issues the correct `LOAD_DEREF` instruction for the read operation.

PYTHON INTERNALS: INTERPRETER

Important terms for the Python interpreter:

- The evaluation loop (or REPL in the interactive Python shell) will take a *code object* and convert it into a series of *frame objects*.
- Frame objects are executed in a so called *frame stack* (what we saw in pythontutor).
- The interpreter manages referenced variables in a *value stack*.
- The interpreter has at least one thread but *at most one thread can run at a time*. The Python interpreter uses an internal global interpreter lock (called GIL) which prevents race conditions and ensures thread safety. The GIL imposes a very strong constraint on multi-threaded execution and was subject to many discussions in the past. A recent post (10/07/2021) on the `python-dev` mailing list proposes a new design to remove the GIL which would mean a major change in the Python interpreter, a possible change that will be reality in the next Python 4 release.
- ***Did you know:*** for the first time in 20 years, Python became the worlds most popular programming language this year → *this was for 2021, Python still is #1 in 2022.*

PYTHON INTERNALS: BACK TO OBJECTS

Everything in Python is an object!

Fixed size object base:

```
typedef struct _object {
    _PyObject_HEAD_EXTRA
    Py_ssize_t ob_refcnt;
    PyTypeObject *ob_type;
} PyObject; // C code in cpython
```

- `_PyObject_HEAD_EXTRA` is a macro that is usually empty.
- `ob_refcnt` is the *reference* count for the object.
- `ob_type` is a pointer to the type object. Recall that Python is *dynamically typed*.

Variable size object base:

```
typedef struct {
    PyObject ob_base;
    Py_ssize_t ob_size;
} PyVarObject; // C code in cpython
```

- `ob_base` is a fixed size object instance.
- `ob_size` is the number of items in the variable part.
- Containers (e.g. `list`) are objects of this type.

No object in Python is a direct instance of `PyObject`. However, every object in Python can be *cast to a `PyObject`* (if it is variable size it can be cast to `PyVarObject` in addition).

PYTHON INTERNALS: FRAME OBJECTS

A frame object is a `PyObject` with the following additional properties:

The `PyFrameObject`:

```c
struct _frame {
    PyObject ob_base;
    struct _frame *f_back;
    struct _interpreter_frame *f_frame;
    PyObject *f_trace;
    int f_lineno;
    char f_trace_lines;
    char f_trace_opcodes;
    char f_own_locals_memory;
} PyFrameObject;
```

- `ob_base` is the base instance (as before).
- `f_back` is a pointer to the previous `PyFrameObject` towards the caller (enables the frame stack).
- `f_frame` is a pointer to the frame data.
- Other fields are used for debugging.

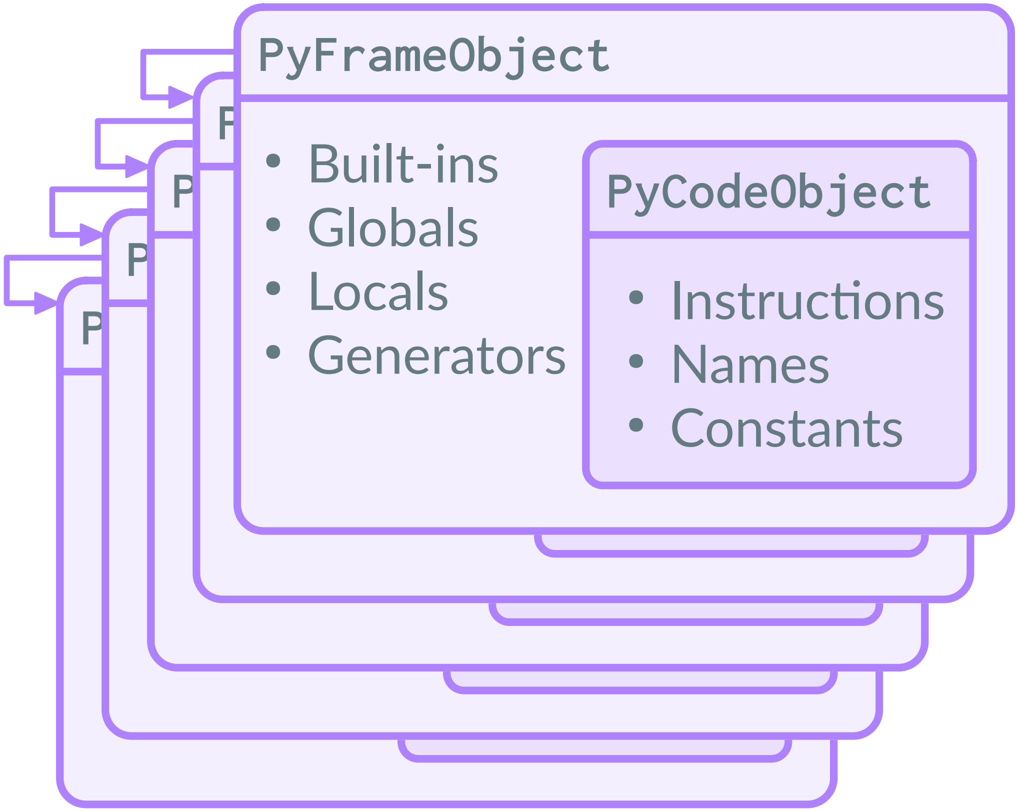
The frame data (some code not shown):

```c
typedef struct _interpreter_frame {
    PyObject *f_globals;
    PyObject *f_builtins;
    PyObject *f_locals;
    PyCodeObject *f_code;
    PyFrameObject *frame_obj;
    PyObject *generator;
    int f_lasti;
    int depth;
} InterpreterFrame;
```

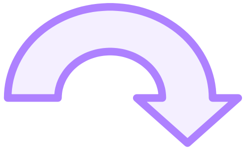
- *Is not an object* (has no `ob_base`)!
- `f_globals` and `f_locals` point to data.
- `f_code` is the bytecode object that will be executed by the frame.
- `f_lasti` index of the last instruction executed → ***Where is this index used?***

PYTHON INTERNALS: FRAME OBJECTS

Frame Stack



Evaluation Loop

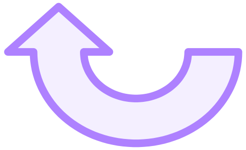


`_PyEval_EvalCode`

- Creates new frame objects from code objects and push/pop them onto the frame stack

`_PyEval_EvalFrameDefault`

- Evaluates the code in frame objects associated with the corresponding values



https://github.com/python/cpython

PYTHON INTERNALS: GENERATOR OBJECTS

The `PyGenObject` (some code not shown):

```
typedef struct {
    /* The gi_ prefix is intended to
    remind of generator-iterator. */
    PyObject ob_base;
    /* Note: gi_frame can be NULL if
    the generator is "finished" */
    struct _interpreter_frame *gi_xframe;
    /* The code object backing
    the generator */
    PyCodeObject *gi_code;
} PyGenObject;
```

- `gi_xframe` points to the current frame object for the generator.
- `gi_code` bytecompiled code object of the *generator function*.
- `PyGenObject`'s are flagged when created with `CO_GENERATOR` (last time), `CO_COROUTINE` (PEP 492) or `CO_ASYNC_GENERATOR` (PEP 525).

- Frame objects have a *pointer to generator objects* and they also store the index of the *last instruction* in the bytecode of the frame.
- The pointer allows to *resume* a generator object that is associated with a frame. The frame data has this code:

```
PyObject *generator;
```

  This *pointer* points to a `PyGenObject` *somewhere* in dynamic memory (*not on the frame stack*).
- Although the frame object is in the frame stack, the `generator` pointer allows to obtain the generator object from somewhere else in memory such that it can be resumed (for example when the frame calls `next()` on it).
- ***Fact:*** Python's frame stack is maintained in *dynamic memory* (*heap*) → ***this does not apply to normal program execution.***

PYTHON INTERNALS: DYNAMIC MEMORY

- Processes share CPU and memory among each other.
- Sharing memory is a non-trivial task when designing operating systems.
- Each physical memory cell (byte-sized cells) can be addressed uniquely. For example:
  - ***32-bit system:*** 4294967296 addresses; can handle *4GB* (gigabyte) of RAM at most.
  - ***64-bit system:*** 18446744073709551616 addresses; can handle *16EB* (exabyte) of RAM at most. (This is *A LOT!*)
- *Virtual memory* simplifies memory management in operating systems by making processes "think" that their memory space always starts at address `0`. The *virtual address* is then translated to the real physical address by a hardware component called memory management unit (MMU).

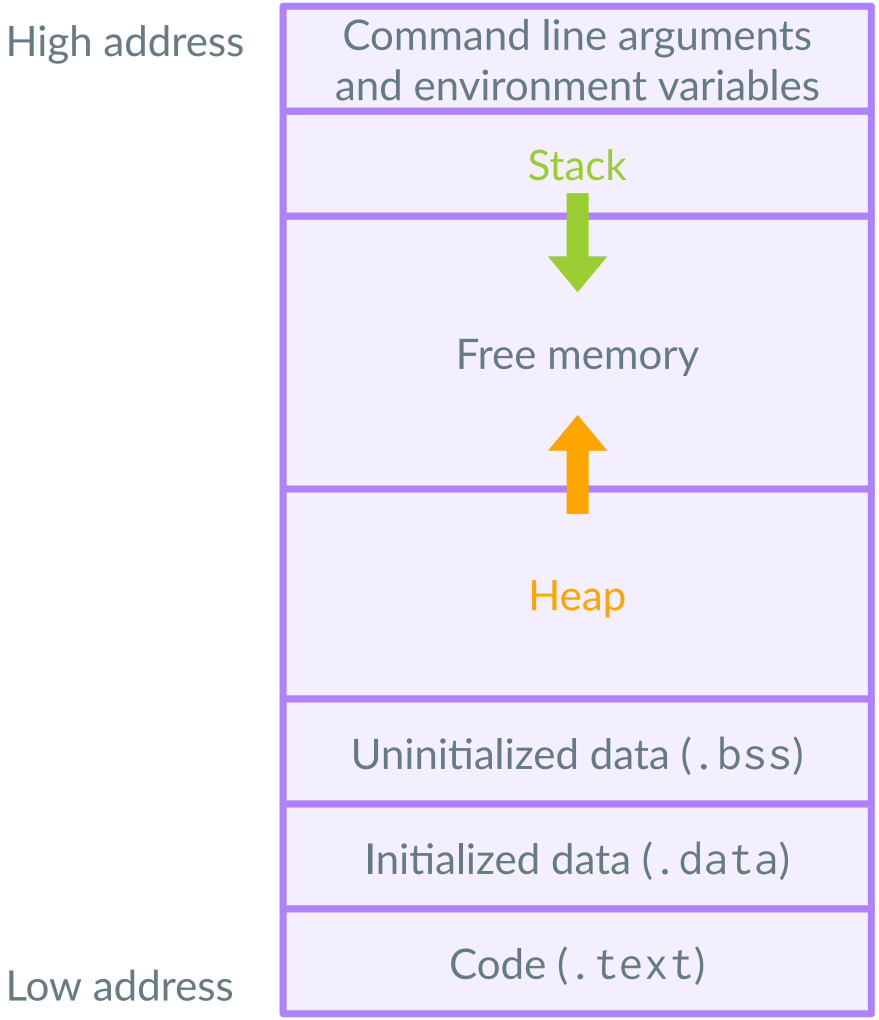
PYTHON INTERNALS: DYNAMIC MEMORY

Different ways (places) to allocate memory:

- ***Static memory allocation:*** variables with known size at compile time. The compiler allocates this memory inside the *executable*.
- ***Automatic memory allocation:*** similar to static memory allocation (the allocation requirements are known at compile time). Allocation is carried out on the stack when the code executes (for example a function body).
- ***Dynamic memory allocation:*** when it is not possible for the compiler to determine a specific memory request the memory must be allocated dynamically. A dynamic memory segment is allocated on the heap. ***Example:*** the allocation size depends on user input.
- All objects in Python (including frame objects and code objects) are *allocated dynamically on the heap* → *a reason why Python is slow!*

PYTHON INTERNALS: DYNAMIC MEMORY

Virtual memory of a Linux process:



LIFO stack (Last-In-First-Out), allocated by the OS at program start. Stack frames are created here (e.g. function calls). Easy to manage and much faster than memory allocation on the heap.

Automatic memory allocation

Stack and heap grow in opposite direction.

Dynamic memory that grows depending on program. Hard to manage (fragmentation, garbage collection) and involves expensive kernel calls (e.g. malloc or new). Allocating memory on the heap is slow.

Dynamic memory allocation

Global and static variables. The linker allocates memory in these segments. Those variables can be either initialized or uninitialized.

Executable machine code (instructions), read-only segment.

Static memory allocation

PYTHON INTERNALS: DYNAMIC MEMORY

- The Python interpreter ***emulates*** a frame stack using ***dynamic memory***. Frame objects are *pushed* and *popped* to and from the frame stack on the heap (dynamic memory pool).
- These "stack" operations are more expensive than true operations on a stack with ***automatic memory allocation*** as it is the case for a `x86_64` executable for example.
- Since all Python objects are allocated on the heap, *generator objects persist* until the interpreter explicitly removes them from the heap. This allows to easily resume a suspended generator including its state. Because a `PyFrameObject` stores the last instruction in `f_lasti`, it will be used to index into the bytecode of a generator object to resume execution with instruction `f_lasti + 1` → that is, after the last active `yield` statement.

PYTHON INTERNALS: SUMMARY

- The Python interpreter exposes a number of internal objects to the user of which we have discussed three:
  - Code objects for byte compiled code
  - Frame objects to execute code.
  - Generator objects for suspension and resumption of code execution.
  - Traceback objects for debugging (not discussed).
- It is rare that you will need to manipulate these objects directly in your Python code.
- We have discussed them here to understand the low-level Python internals without going too deep into the interpreter source code.
- The most important source code of the CPython interpreter is `ceval.c`.

PYTHON LIST OBJECTS AND NUMPY ARRAYS

- We have seen that excessive use of dynamic memory allocation in Python is a cause for its reputation of being slow.
- It is not due to bad design (the GIL is debatable), but rather the cost of prioritizing *flexibility* and the possibility for *fast prototyping* which has value in its own right.
- One of the reasons for this performance penalty is that Python objects are not necessarily near by in memory due to dynamic memory allocation and object oriented design.
- ***Why does this matter since memory cells in random access memory (RAM) can be accessed in constant time you may ask?***
  - Additional pointer dereferences until you get to the data → *in Python everything is an object and must be referenced by pointers*.
  - *Spatial and temporal locality* of the data is not optimal. Results in many cache misses when reading or writing data.

PYTHON LIST OBJECTS AND NUMPY ARRAYS

Python built-in lists:

- Let us see how list objects are implemented in Python:

```c
typedef struct {
    PyVarObject ob_base;
    // Vector of pointers to
    // list elements.  list[0]
    // is ob_item[0], etc.
    PyObject **ob_item;
    Py_ssize_t allocated;
} PyListObject;
```

- `ob_item` is *a pointer to pointer(s)* to `PyObject`'s. ***Example:*** `ob_item[0]` returns a pointer to a `PyObject`, `ob_item[1]` returns the next pointer to the second `PyObject` and so on.

- In the following we assume `PyObject` represents a Python integer:

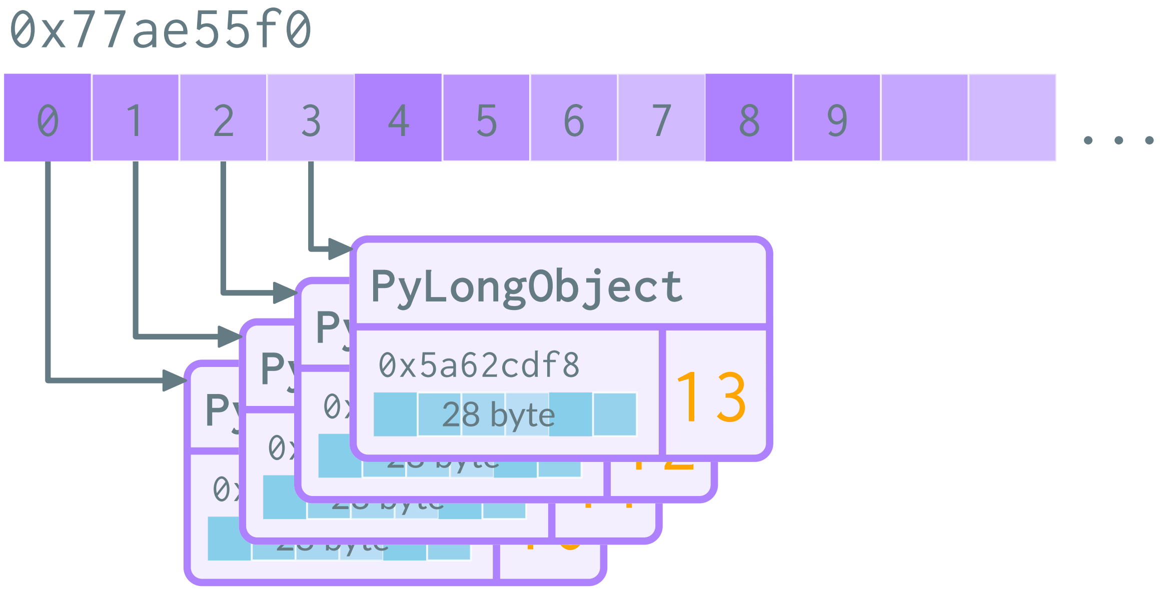
```c
struct _longobject {
    PyObject ob_base;
    Py_ssize_t ob_size;
    // the actual integer:
    digit ob_digit[1];
} PyLongObject;
```

We assume that the `PyObject` takes *16 byte*, `ob_size` is *8 byte* and `ob_digit` is *4 byte*. A `PyLongObject` then has a size of *28 byte*.

- The actual integer value is stored at `ob_digit[1]`.

PYTHON LIST OBJECTS AND NUMPY ARRAYS

Python built-in lists:

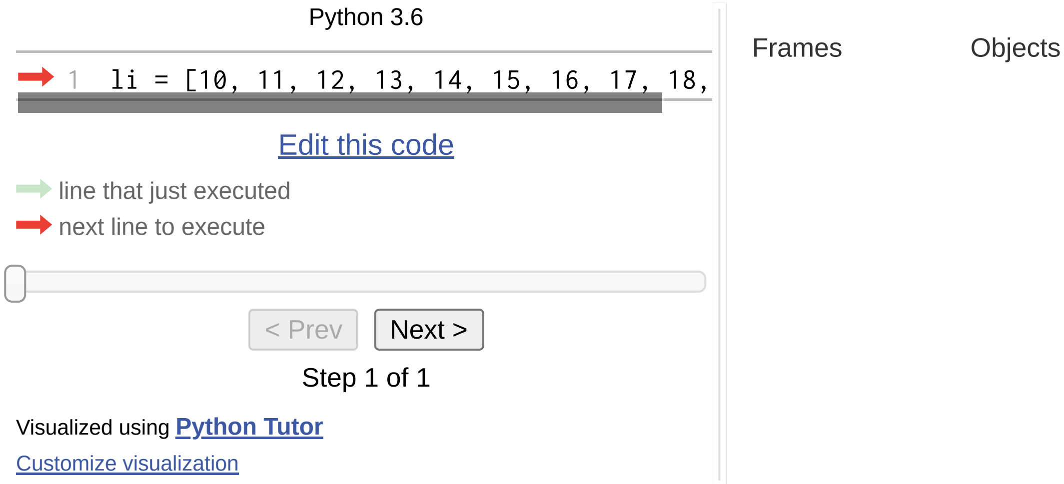



```
1  li = [10, 11, 12, 13, 14, 15, 16, 17, 18, 19]  # assume this list of integers
```

- The elements of `ob_item` are *coalesced* in memory. We can access the `PyObject` references in the list with $\mathcal{O}(1)$ complexity.
- To obtain the actual value we must *dereference* the pointer and read `ob_digit[0]` for every item in the list!

PYTHON LIST OBJECTS AND NUMPY ARRAYS

We can visualize this list on pythontutor.com showing all heap allocations:



PYTHON LIST OBJECTS AND NUMPY ARRAYS

Example: `for`-loop over iterable and sum values

Sum values in iterable x:

```python
import timeit
import numpy as np

def pysum(x):
    s = 0
    for i in x:
        s += i

if __name__ == "__main__":
    x = np.array(list(range(1000000)))
    t = timeit.timeit('f(x)',
        globals={'f': pysum, 'x': x},
        number=10)
```

Instructions for pysum code object:

```
5           0 LOAD_CONST               1 (0)
            2 STORE_FAST               1 (s)

6           4 LOAD_FAST                0 (x)
            6 GET_ITER
      >>    8 FOR_ITER                12 (to 22)
           10 STORE_FAST               2 (i)

7          12 LOAD_FAST                1 (s)
           14 LOAD_FAST                2 (i)
           16 INPLACE_ADD
           18 STORE_FAST               1 (s)
           20 JUMP_ABSOLUTE            8
      >>   22 LOAD_CONST               0 (None)
           24 RETURN_VALUE
```

Running this code with 1'000'000 elements takes ***0.74 seconds***, averaged over 10 samples.

PYTHON LIST OBJECTS AND NUMPY ARRAYS

- While `PyObject`'s in Python are very flexible (Python is dynamically typed), this flexibility comes at a performance price.
- The Python interpreter is designed to work with `PyObject`'s exclusively → *everything in Python is an object*.
- Performance oriented designs are centered around ***data*** rather than objects.
- Because the Python interpreter is written in C, extensions can easily be implemented.
- NumPy is a Python extension module designed for efficient numerical computation in Python.

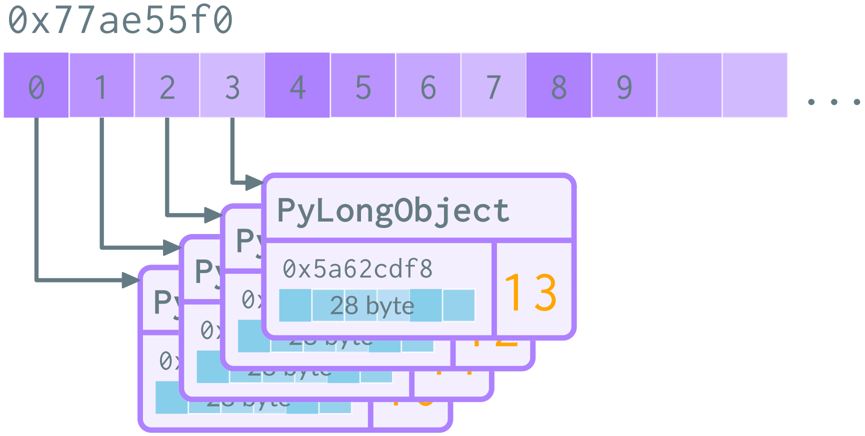
  It operates on its own data structures in order not to pay the performance price for flexibility → `np.array`

PYTHON LIST OBJECTS AND NUMPY ARRAYS

Back to our list:

```python
li = [10, 11, 12, 13, 14, 15, 16, 17, 18, 19]  # assume this list of integers
```

Object oriented Python list:

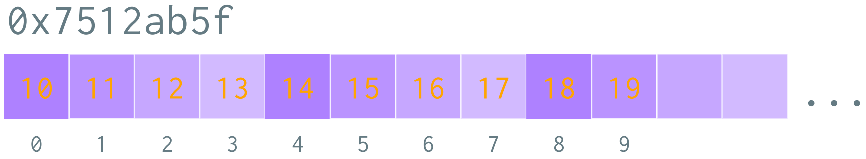


- The list above is the same as

```c
// 10 contiguous pointer
PyObject *ob_item[10];
```

- `PyLongObject` uses 32-bit integral type for integers, i.e., `int` in C.

Data oriented NumPy array:



- The elements of a NumPy array are contiguous data items, not pointers (references) to `PyObject`'s.
- The NumPy array above is similar to

```c
// 10 contiguous data items
int ob_item[10];
```

- ***Reading the data in this format will saturate the memory bandwidth!***

PYTHON LIST OBJECTS AND NUMPY ARRAYS

Example: `for`-loop over iterable and sum values (as before)

Sum values in iterable x:

```python
import timeit
import numpy as np

def pysum(x):
    s = 0
    for i in x:
        s += i

def npsum(x):
    s = x.sum()

if __name__ == "__main__":
    x = np.array(list(range(1000000)))
    t = timeit.timeit('f(x)',
        globals={'f': npsum, 'x': x},
        number=10)
```

Instructions for npsum code object:

```
10       0 LOAD_FAST            0 (x)
         2 LOAD_METHOD          0 (sum)
         4 CALL_METHOD          0
         6 STORE_FAST           1 (s)
         8 LOAD_CONST           0 (None)
        10 RETURN_VALUE
```

Running this code with 1'000'000 elements averaged over 10 samples:

- Pure Python: ***0.74 seconds***
- NumPy array: ***0.0046 seconds***

→ ***Two orders of magnitude faster!***

Note: the `sum()` built-in function is only slightly faster (***0.60 seconds***) than the naive `for`-loop implementation!

RECAP

- Python internals:
  - Code objects and bytecode
  - The interpreter and the evaluation loop
  - Frame objects
  - Generator objects
- Why are Python built-in lists slow and NumPy arrays fast?

Further reading:

- Python bytecode disassembler: https://docs.python.org/3/library/dis.html
- CPython interpreter source code: https://github.com/python/cpython
- Python/C API Reference Manual: https://docs.python.org/3/c-api/index.html
- Chapters "The Evaluation Loop", "Memeory Management" and "Objects and Types" in Anthony Shaw, *"CPython Internals: Your Guide to the Python 3 Interpreter"*, Real Python, 2020