CS107 / AC207

SYSTEMS DEVELOPMENT FOR COMPUTATIONAL SCIENCE

LECTURE 19

Tuesday, November 9th 2021

Fabian Wermelinger
Harvard University

RECAP OF LAST TIME

- Generators
- Coroutines
- `python` internals: objects, bytecode and interpreter

OUTLINE

- `python` internals:
  - Code objects
  - The interpreter and the evaluation loop
  - Frame objects
  - Generator objects
  - Traceback objects
- `python` lists and `numpy` arrays

`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 (the *left column* you saw in pythontutor).
- What is still missing to execute these frame objects is *input* data (the *right column* you saw in pythontutor).
- The `python` 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 `python` interpreter is written in `C` (you can inspect the source code at 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.*

python INTERNALS: CODE OBJECTS

- Code objects is what the `python` interpreter executes. 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'
```

- 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

- 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):
...     l = x
...     print(f'f() local variables:  {locals()}')
...     print(f'f() global variables: {globals()}')
...
>>> g = 0
>>> f(g)
f() local variables:  {'x': 0, 'l': 0}
f() global variables: {'__name__': '__main__', '__doc__': None, '__package__': None,
'__loader__': <class '_frozen_importlib.BuiltinImporter'>, '__spec__': None,
'__annotations__': {}, '__builtins__': <module 'builtins' (built-in)>,
'f': <function f at 0x7f6868f8f280>, 'g': 0}
```

python INTERNALS: INTERPRETER

Short detour: low-level explanation of two things we have seen previously in the lecture: 1.) the `nonlocal` keyword

```python
def f(x):
    def g(y):
        z = x + y
        return z
    return g
```

```python
>>> g = f(0)
>>> f.__code__.co_cellvars # tuple of vars referenced in nested functions
('x',)
>>> g.__closure__
(<cell at 0x7f90da60fc10: int object at 0x7f90da769910>,)
>>> g.__closure__[0].cell_contents
0
>>> g(1)
1
>>> g.__closure__[0].cell_contents
0
```

Read-only `x` !

```python
1 def f(x):
2     def g(y):
3         z = x + y
4         x = y
5         return z
6     return g
```

```python
>>> g = f(0)
>>> f.__code__.co_cellvars # tuple of vars referenced in nested functions
()
>>> g.__closure__ # is None -- no cellvars in code object this time
>>> g(1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in g
UnboundLocalError: local variable 'x' referenced before assignment
>>> import dis; dis.dis(g)
  3           0 LOAD_FAST                1 (x)
              2 LOAD_FAST                0 (y)
              4 BINARY_ADD
              6 STORE_FAST               2 (z)
... # more assembly ignored
```

The `UnboundLocalError` happens in `line 3` (first access) because it is not defined in the scope!

python INTERNALS: INTERPRETER

Short detour: low-level explanation of two things we have seen previously in the lecture: 1.) the `nonlocal` keyword

```python
def f(x):
    def g(y):
        z = x + y
        return z
    return g
```

Read-only `x` !

```
>>> g = f(0)
>>> f.__code__.co_cellvars # tuple of vars referenced in nested functions
('x',)
>>> g.__closure__
(<cell at 0x7f90da60fc10: int object at 0x7f90da769910>,)
>>> g.__closure__[0].cell_contents
0
>>> g(1)
1
>>> g.__closure__[0].cell_contents
0
```

```python
def f(x):
    def g(y):
        x = y
        z = x + y
        return z
    return g
```

Will this result in an `UnboundLocalError` ?

python INTERNALS: INTERPRETER

Short detour: low-level explanation of two things we have seen previously in the lecture: 1.) the `nonlocal` keyword

```python
def f(x):
    def g(y):
        nonlocal x
        z = x + y
        x = y
        return z
    return g
```

```
>>> g = f(0)
>>> f.__code__.co_cellvars # tuple of vars referenced in nested functions
('x',)
>>> g.__closure__
(<cell at 0x7f7bdcf37a60: int object at 0x7f7bdd0a2910>,)
>>> g.__closure__[0].cell_contents
0
>>> g(1)
1
>>> g.__closure__[0].cell_contents
1
```

Read-Write captured `x` !

```python
1 def f(x):
2     def g(y):
3         z = x + y
4         x = y
5         return z
6     return g
```

```
>>> g = f(0)
>>> f.__code__.co_cellvars # tuple of vars referenced in nested functions
()
>>> g.__closure__ # is None -- no cellvars in code object this time
>>> g(1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in g
UnboundLocalError: local variable 'x' referenced before assignment
>>> import dis; dis.dis(g)
  3           0 LOAD_FAST                1 (x)
              2 LOAD_FAST                0 (y)
              4 BINARY_ADD
              6 STORE_FAST               2 (z)
... # more assembly ignored
```

The `UnboundLocalError` happens in `line 3` (first access) because it is not defined in the scope!

python INTERNALS: INTERPRETER

Short detour: low-level explanation of two things we have seen previously in the lecture: 2.) positional and keyword only arguments

```
>>> def f(posonly, /, pos_or_kw=None, *, kwonly=None):
...     pass
...
>>> f.__code__.co_argcount
2
>>> f.__code__.co_posonlyargcount
1
>>> f.__code__.co_kwonlyargcount
1
```

- The compiled bytecode is aware of what arguments are expected.
- `co_argcount` does not include keyword-only arguments (https://docs.python.org/3/library/inspect.html#types-and-members)
- See PEP 457 and PEP 570 for the introduction of positional-only arguments and the control flow tutorial for more discussion.

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.

python INTERNALS: BACK TO OBJECTS

Repeat: everything in `python` is an object!

Fixed size object base:

```c
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:

```c
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`. BUT, 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.

python INTERNALS: FRAME OBJECTS

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

The `PyFrameObject`:

```
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.

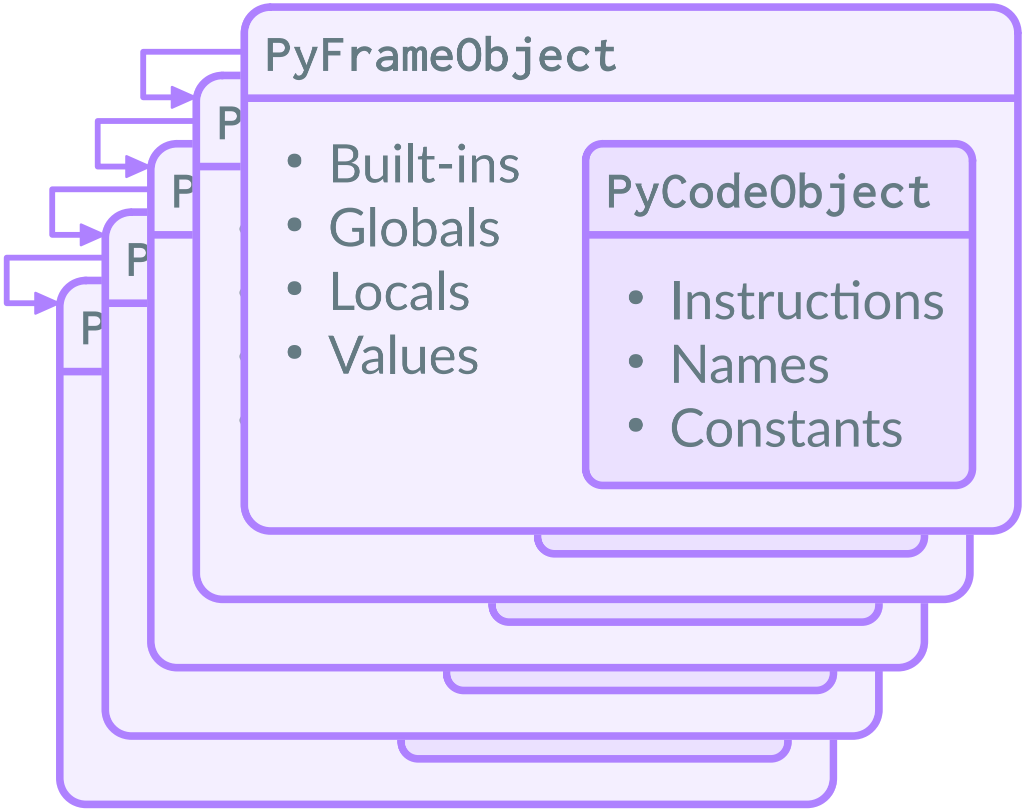
The frame data (some code not shown):

```
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;
```

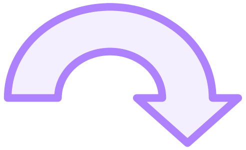
- *Is not an object* (has no `ob_base`). `f_globals` and `f_locals` point to data.
- `f_code` is the bytecode object that will be executed by the frame.
- `f_lasti` index of the last instruction executed. ***Where is this index used?***

python INTERNALS: FRAME OBJECTS

Frame Stack



Evaluation Loop

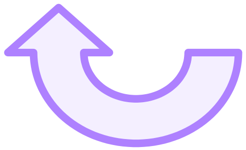


`_PyEval_EvalCode`

- 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):

```c
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` (this class), `CO_COROUTINE` (PEP 492) or `CO_ASYNC_GENERATOR` (PEP 525).

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

```c
PyObject *generator;
```

This is a *pointer* to a `PyObject` "somewhere" in memory. ***Aside:*** pointers in `C/C++` are used for *dynamic memory management*.

- Although the frame object is in the stack, the `generator` pointer allows to obtain the generator object from somewhere else in memory which can then be resumed (e.g. the frame evaluates `next()` on a generator).
- In fact: `python`'s frame stack is maintained in *dynamic memory* (*heap*), something that is not true for standard 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).

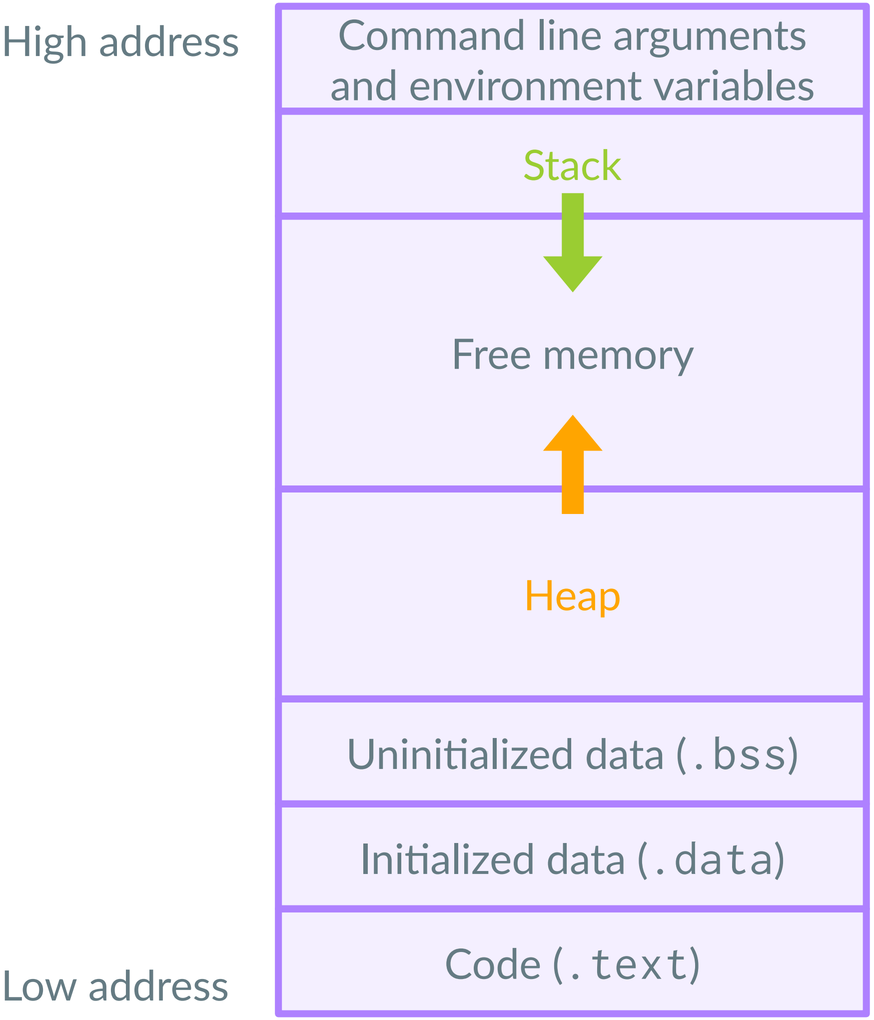
`python` INTERNALS: DYNAMIC MEMORY

- ***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.
- ***Dynamic memory allocation:*** when it is not possible for the compiler to determine a specific memory request, e.g. the allocation size depends on user input at runtime, the memory must be allocated dynamically. A dynamic memory segment is allocated on the heap.

  All objects in `python` (including frame objects and code objects) are *allocated dynamically on the heap*.

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.

Static memory allocation

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

`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 the ones used with ***automatic memory allocation*** in a `x86_64` executable.
- 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`.

`python` INTERNALS: TRACEBACK OBJECTS

- The `python` interpreter exposes a number of internal objects to the user of which we have discussed three so far:
  - Code objects for byte compiled code
  - Frame objects to execute code.
  - Generator objects for suspension and resumption of code execution.
- It is rare that you will need to manipulate these objects directly in your code. We have used them here to understand the low-level `python` internals without going too deep into the interpreter source code.
- The last `python` internal object we want to look at are *traceback objects*. They are created when *exceptions* are raised and used for debugging purposes or non-standard exception handling.

python INTERNALS: TRACEBACK OBJECTS

Obtain a traceback object from an exception:

```python
import dis

def g():
    raise Exception

def f():
    g()

def main():
    try:
        f()
    except Exception as e:
        tb = e.__traceback__
        i = 0
        while tb is not None:
            frame = tb.tb_frame
            li = frame.f_code.co_code[frame.f_lasti]
            print(f'frame {i}: line frame={frame.f_lineno}; ' +
                  f'line trace={tb.tb_lineno}; ' +
                  f'last instruction={dis.opname[li]}')
            tb = tb.tb_next
            i += 1
```

Output:

```
frame 0: line frame=18; line trace=11; last instruction=LOAD_ATTR
frame 1: line frame=7; line trace=7; last instruction=CALL_FUNCTION
frame 2: line frame=4; line trace=4; last instruction=RAISE_VARARGS
```

- Traceback objects can only be obtained through an exception.
- They are similar to frame objects, except that `tb_next` points *towards the frame where the exception has been thrown*.
- Control on the left is in the `main()` function after the exception was thrown. The line number in the frame and trace objects are different because we execute this frame.

python INTERNALS: TRACEBACK OBJECTS

Standard traceback:

```python
def g():
    raise Exception("A useful description goes here")

def f():
    g()

def main():
    f()

if __name__ == "__main__":
    main()
```

Output:

```
Traceback (most recent call last):
  File "/home/fabs/CS107/traceback/tb.py", line 17, in <module>
    main()
  File "/home/fabs/CS107/traceback/tb.py", line 13, in main
    f()
  File "/home/fabs/CS107/traceback/tb.py", line 9, in f
    g()
  File "/home/fabs/CS107/traceback/tb.py", line 5, in g
    raise Exception("A useful description goes here")
Exception: A useful description goes here
```

- Read `python` traceback from bottom up.
- Each line that starts with `File` corresponds to a new traceback object.
- The defaults contain information about the file where the code is executed, the current line and the name of the frame.
- Recall that code blocks in `python` are among the following: modules, function bodies, class definitions or commands typed interactively.

`python` LIST OBJECTS AND NUMPY ARRAYS

- `python` has the reputation of being slow.
- This is true. It is not due to bad design however (the GIL is debatable), but rather prioritizing flexibility and the possibility for fast prototyping.
- One of the reasons for this performance penalty is that `python` objects are not necessarily near by in memory due to dynamic memory allocation.
- How does this matter since memory cells in random access memory (RAM) can be accessed in constant time?
  - Additional pointer dereferences until you get to the data. (Everything 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

- Let us see how list objects are implemented in `python`:

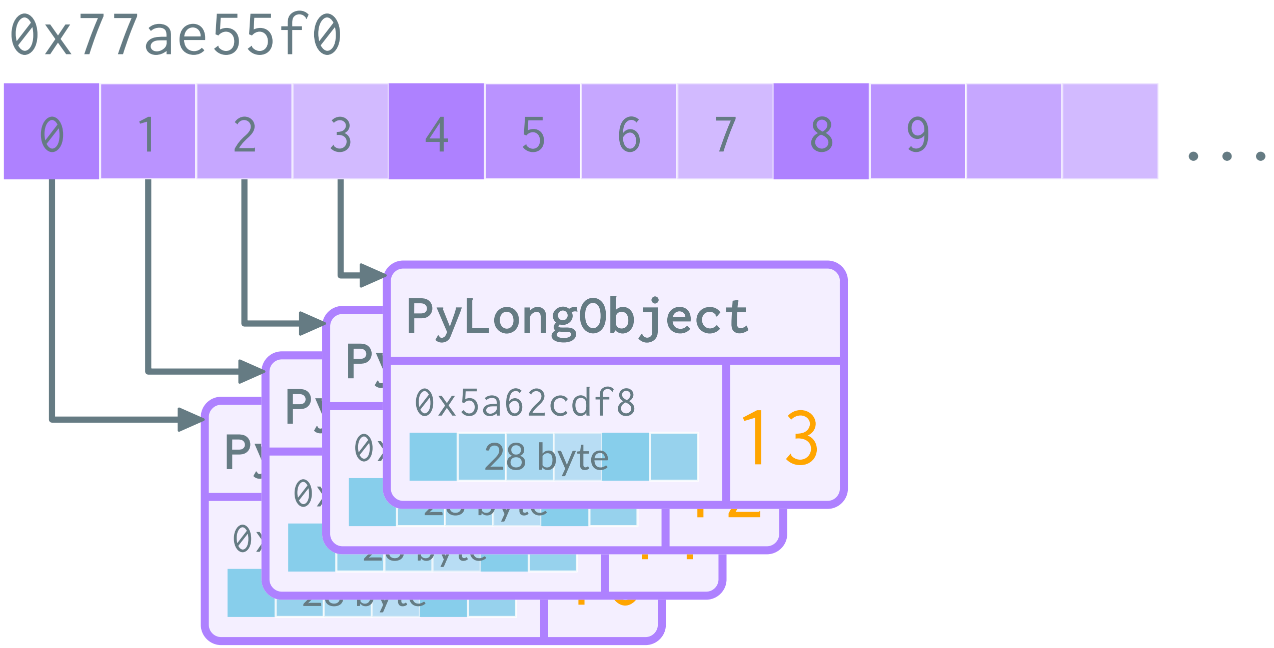
```c
typedef struct {
    PyVarObject ob_base;
    // Vector of pointers to list elements.  list[0] is ob_item[0], etc.
    PyObject **ob_item;
    Py_ssize_t allocated;
} PyListObject;
```

- `ob_item` is *a pointer* to pointer(s) to `PyObject`'s. For 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;
    digit ob_digit[1]; // the actual integer
} 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.

python LIST OBJECTS AND NUMPY ARRAYS



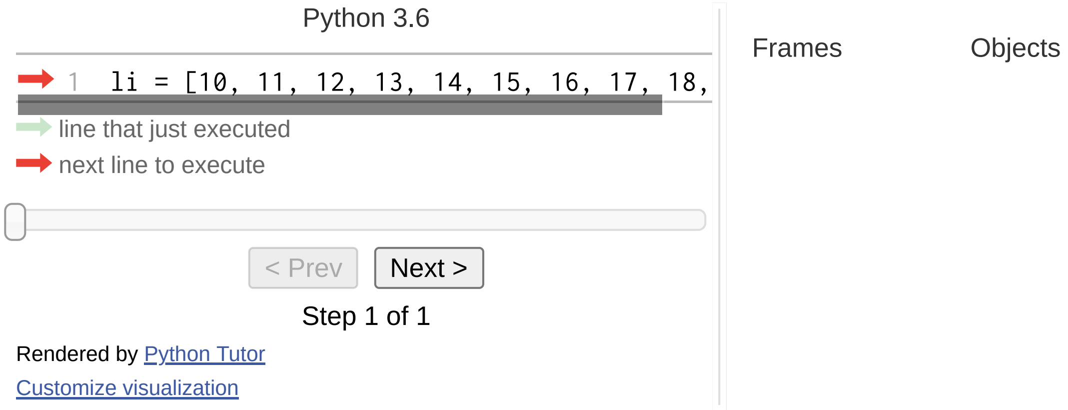
- Assume the following `python` list

```
li = [10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
```

- 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

Sum values in iterable `x`:

```python
import timeit
import numpy as np

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

def main():
    x = np.array(list(range(1000000)))
    t = timeit.timeit('f(x)',
        globals={'f': pysum, 'x': x},
        number=10
    )

if __name__ == "__main__":
    main()
```

Assembly:

```
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 generic in the sense that they are very flexible in terms of describing data, the generality comes at a performance price.
- The `python` interpreter is designed to work with `PyObject`'s exclusively (***recall:*** 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 for this reason.

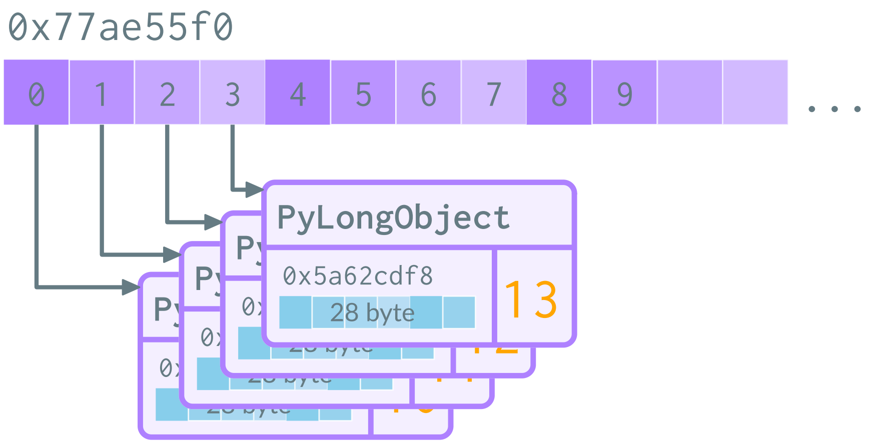
python LIST OBJECTS AND NUMPY ARRAYS

Back to this list:

```
li = [10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
```

Object oriented

python list:



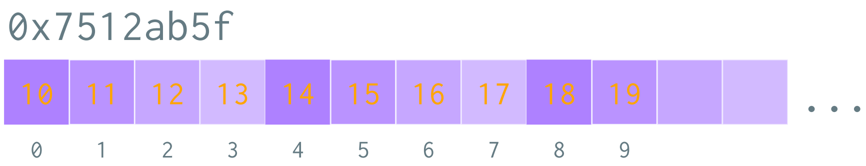
- The list above is the same as

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

- 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

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

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

python LIST OBJECTS AND NUMPY ARRAYS

Example: `for`-loop over iterable (same 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()

def main():
    x = np.array(list(range(1000000)))
    t = timeit.timeit('f(x)',
        globals={'f': npsum, 'x': x},
        number=10
    )

if __name__ == "__main__":
    main()
```

Assembly:

```
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
  - The interpreter and the evaluation loop
  - Frame objects
  - Generator objects
  - Traceback objects
- `python` lists and `numpy` arrays