CPython Memory Structure

Janis Lesinskis, Alysha Iannetta

Where objects are stored and what those objects look like in memory. This will help explain the underpinnings of the way in which equality and identity checks work in CPython and also how Python objects get stored in RAM.

Built-in Objects

The built-in keywords are PyObjects but are stored in a special spot. There's a fixed sized area set aside for built-in objects.
Values such as **True**, **False** and **None** are stored here. They are all stored as singletons.
This means that any identity comparison will always be true for these particular objects.
For example any variable that has the value of **None** will always point to the same address. They will have the same id as there is only one **None** object which lives in that special area of memory.

```
t = True
t1 = True
n = None
n1 = None
```

```
id(t)
# 1916989600
id(t1)
# 1916989600
```

```
id(n)
# 1917035664
id(n1)
# 1917035664
```

Identity vs Equality

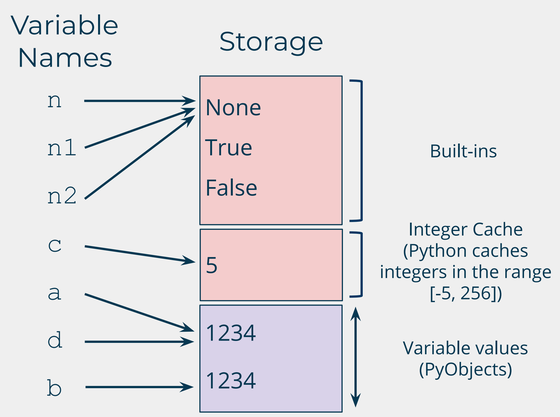
- == checks if two variables have equal contents.
- **is** checks if two variable names are the same variable (same spot in storage space).

```
n = None
n1 = None
n2 = None

a = 1234
b = 1234
c = 5
d = a
```

```
n is n1
# True
n == n1
# True
n1 is n2
# True
n1 == n2
# True
```

```
a is b
# False
a == b
# True
d is a
# True
d == a
# True
```

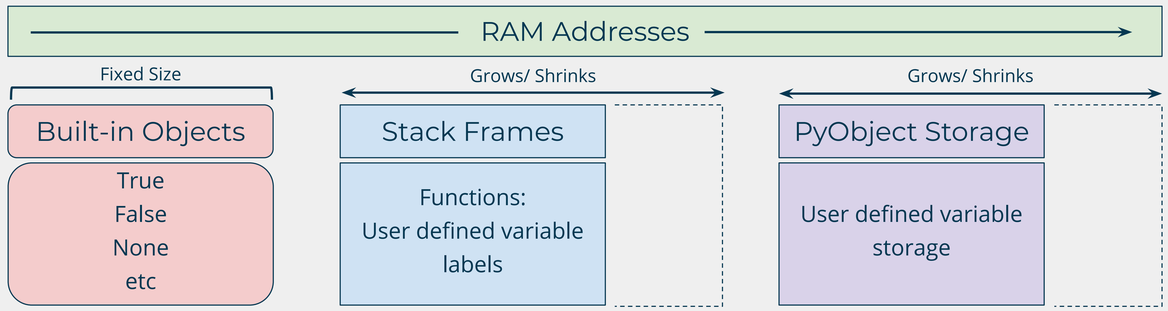


CPython Objects - Where everything goes

First some assumptions:

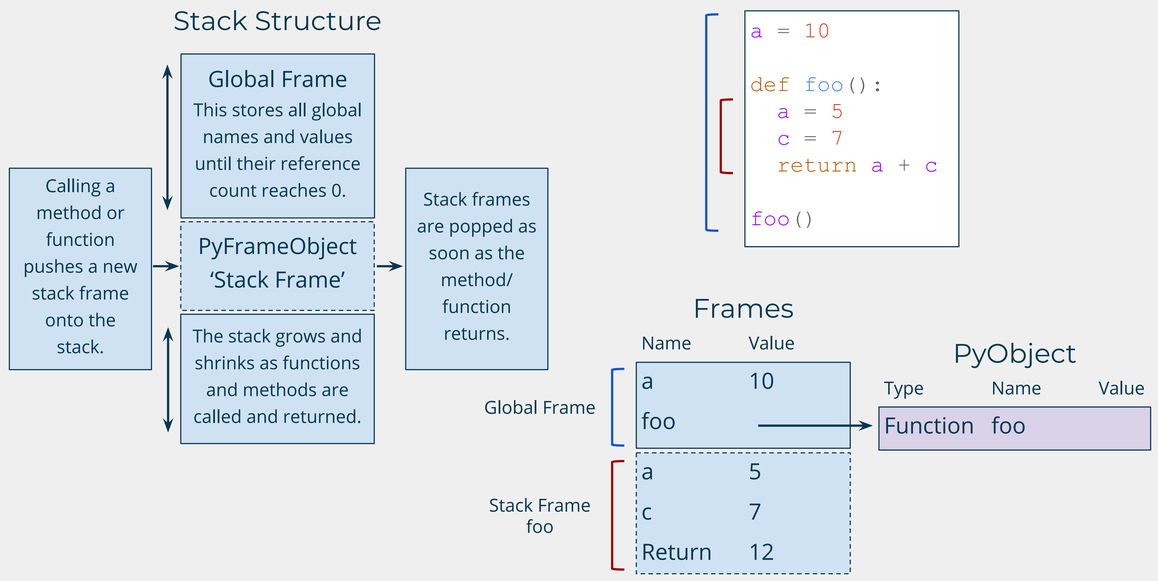
- We are going to assume that RAM is addressable in a sequential manner.
- We are also going to assume that a given address is a unique location that we can store information at. Or alternatively that any two distinct addresses are distinct locations in memory.

When you start the CPython interpreter it sets up the variables and memory that the interpreter needs to run your code. It generates this memory layout:



Stack Frames

Stack frames contain all the information required to allow function calls to work.
Functions are themselves variables but they have a few extra things going on too. Specifically they need to be able to set aside some memory for storing the references to the other variables stored in them.
When we create a function we have to put aside the memory for all the variables inside it as well as the function call machinery.
This is most easily achieved with a stack based structure and therefore is how CPython has implemented it. These entities are called frame objects.



PyObject

Python is an "Object Oriented Language", which in this case means that almost every variable is accessible via a similar interface.
Python has the notion of a data model where objects are the abstraction Python uses for data, and as many things as possible are stored in these objects as possible across the language.
These are PyObjects in the source and if you look around the source code you'll see these PyObjects all over the place.

What is a PyObject? From the source (cpython/Include/object.h):

```
typedef struct _object {
   _PyObject_HEAD_EXTRA
   Py_ssize_t ob_refcnt;
   struct _typeobject *ob_type;
} PyObject;

#define _PyObject_CAST(op) ((PyObject*)(op))

typedef struct {
   PyObject ob_base;
   Py_ssize_t ob_size;
} PyVarObject;
```

- This gives us the base from which everything else is built.
- This is a placeholder into which we can store arbitrary variables.

PyObject

`ob_refcnt`
This provides for the reference counted memory management of anything stored within it.

`*ob_type`
A pointer to a type object that indicates the type of an object. Everything in Python must have information about the type, otherwise Python wouldn't know how to handle these entities.

You can find the type of any item in Python using **type**:

```
type('abc')
# <class 'str'>
type([1,2,3])
# <class 'list'>
```

VarPyObject

`PyObject ob_base`
Extends on PyObject.

`Py_ssize_t ob_size`
Used to count size of objects that have some notion of length.

PYTHON CHARMERS

Custom Programming Solutions