merge2

June 14, 2023

0.1 Variables are Memory References

We can find the memory address that a variable *references*, by using the `id()` function.

The `id()` function returns the memory address of its argument as a base-10 integer.

We can use the function `hex()` to convert the base-10 number to base-16.

[1]:
```
my_var = 10
print('my_var = {0}'.format(my_var))
print('memory address of my_var (decimal): {0}'.format(id(my_var)))
print('memory address of my_var (hex): {0}'.format(hex(id(my_var))))
```

```
my_var = 10
memory address of my_var (decimal): 1968827120
memory address of my_var (hex): 0x7559eaf0
```

[2]:
```
greeting = 'Hello'
print('greeting = {0}'.format(greeting))
print('memory address of my_var (decimal): {0}'.format(id(greeting)))
print('memory address of my_var (hex): {0}'.format(hex(id(greeting))))
```

```
greeting = Hello
memory address of my_var (decimal): 1688681719264
memory address of my_var (hex): 0x1892d4625e0
```

Note how the memory address of `my_var` is **different** from that of `greeting`.

Strictly speaking, `my_var` is not "equal" to 10.

Instead `my_var` is a **reference** to an (*integer*) object (*containing the value 10*) located at the memory address `id(my_var)`

Similarly for the variable `greeting`.

0.2 Reference Counting

Method that returns the reference count for a given variable's memory address:

[1]:
```
import ctypes

def ref_count(address):
```

```
    return ctypes.c_long.from_address(address).value
```

Let's make a variable, and check it's reference count:

[2]:
```
my_var = [1, 2, 3, 4]
ref_count(id(my_var))
```

[2]: 1

There is another built-in function we can use to obtain the reference count:

[3]:
```
import sys
sys.getrefcount(my_var)
```

[3]: 2

But why is this returning 2, instead of the expected 1 we obtained with the previous function?

Answer: The *sys.getrefcount()* function takes **my_var** as an argument, this means it receives (and stores) a reference to **my_var**'s memory address **also** - hence the count is off by 1. So we will use *from_address()* instead.

We make another reference to the **same** reference as `my_var`:

[4]:
```
other_var = my_var
```

Let's look at the memory address of those two variables and the reference counts:

[5]:
```
print(hex(id(my_var)), hex(id(other_var)))
print(ref_count(id(my_var)))
```

```
0x1e43f368388 0x1e43f368388
2
```

Force one reference to go away:

[6]:
```
other_var = None
```

And we look at the reference count again:

[7]:
```
print(ref_count(id(my_var)))
```

```
1
```

We see that the reference count has gone back to 1.

You'll probably never need to do anything like this in Python. Memory management is completely transparent - this is just to illustrate some of what is going behind the scenes as it helps to understand upcoming concepts.

0.3 Garbage Collection

[1]:
```
import ctypes
import gc
```

We use the same function that we used in the lesson on reference counting to calculate the number of references to a specified object (using its memory address to avoid creating an extra reference)

[2]:
```
def ref_count(address):
    return ctypes.c_long.from_address(address).value
```

We create a function that will search the objects in the GC for a specified id and tell us if the object was found or not:

[3]:
```
def object_by_id(object_id):
    for obj in gc.get_objects():
        if id(obj) == object_id:
            return "Object exists"
    return "Not found"
```

Next we define two classes that we will use to create a circular reference

Class A's constructor will create an instance of class B and pass itself to class B's constructor that will then store that reference in some instance variable.

[4]:
```
class A:
    def __init__(self):
        self.b = B(self)
        print('A: self: {0}, b:{1}'.format(hex(id(self)), hex(id(self.b))))
```

[5]:
```
class B:
    def __init__(self, a):
        self.a = a
        print('B: self: {0}, a: {1}'.format(hex(id(self)), hex(id(self.a))))
```

We turn off the GC so we can see how reference counts are affected when the GC does not run and when it does (by running it manually).

[6]:
```
gc.disable()
```

Now we create an instance of A, which will, in turn, create an instance of B which will store a reference to the calling A instance.

[7]:
```
my_var = A()
```

```
B: self: 0x1fc1eae44e0, a: 0x1fc1eae4908
A: self: 0x1fc1eae4908, b:0x1fc1eae44e0
```

As we can see A and B's constructors ran, and we also see from the memory addresses that we have a circular reference.

In fact `my_var` is also a reference to the same A instance:

[8]:
```
print(hex(id(my_var)))
```

```
0x1fc1eae4908
```

Another way to see this:

[9]:
```
print('a: \t{0}'.format(hex(id(my_var))))
print('a.b: \t{0}'.format(hex(id(my_var.b))))
print('b.a: \t{0}'.format(hex(id(my_var.b.a))))
```

```
a:      0x1fc1eae4908
a.b:    0x1fc1eae44e0
b.a:    0x1fc1eae4908
```

[10]:
```
a_id = id(my_var)
b_id = id(my_var.b)
```

We can see how many references we have for `a` and `b`:

[11]:
```
print('refcount(a) = {0}'.format(ref_count(a_id)))
print('refcount(b) = {0}'.format(ref_count(b_id)))
print('a: {0}'.format(object_by_id(a_id)))
print('b: {0}'.format(object_by_id(b_id)))
```

```
refcount(a) = 2
refcount(b) = 1
a: Object exists
b: Object exists
```

As we can see the A instance has two references (one from `my_var`, the other from the instance variable `b` in the B instance)

The B instance has one reference (from the A instance variable `a`)

Now, let's remove the reference to the A instance that is being held by `my_var`:

[12]:
```
my_var= None
```

[13]:
```
print('refcount(a) = {0}'.format(ref_count(a_id)))
print('refcount(b) = {0}'.format(ref_count(b_id)))
print('a: {0}'.format(object_by_id(a_id)))
print('b: {0}'.format(object_by_id(b_id)))
```

```
refcount(a) = 1
refcount(b) = 1
a: Object exists
b: Object exists
```

As we can see, the reference counts are now both equal to 1 (a pure circular reference), and reference counting alone did not destroy the A and B instances - they're still around. If no garbage collection is performed this would result in a memory leak.

Let's run the GC manually and re-check whether the objects still exist:

[14]:
```
gc.collect()
print('refcount(a) = {0}'.format(ref_count(a_id)))
print('refcount(b) = {0}'.format(ref_count(b_id)))
print('a: {0}'.format(object_by_id(a_id)))
print('b: {0}'.format(object_by_id(b_id)))
```

```
refcount(a) = 0
refcount(b) = 0
a: Not found
b: Not found
```

0.3.1 Dynamic Typing

Python is dunamically typed.

This means that the type of a variable is simply the type of the object the variable name points to (references). The variable itself has no associated type.

[1]:
```
a = "hello"
```

[2]:
```
type(a)
```

[2]:
```
str
```

[3]:
```
a = 10
```

[4]:
```
type(a)
```

[4]:
```
int
```

[5]:
```
a = lambda x: x**2
```

[6]:
```
a(2)
```

[6]:
```
4
```

[7]:
```
type(a)
```

[7]:
```
function
```

As you can see from the above examples, the type of the variable `a` changed over time - in fact it was simply the type of the object `a` was referencing at that time. No type was ever attached to the variable name itself.

[ ]:

```

```

0.4 Variable Re-Assignment

Notice how the memory address of **a** is different every time.

[1]:
```
a = 10
hex(id(a))
```

[1]: '0x7559eaf0'

[2]:
```
a = 15
hex(id(a))
```

[2]: '0x7559eb90'

[3]:
```
a = 5
hex(id(a))
```

[3]: '0x7559ea50'

[4]:
```
a = a + 1
hex(id(a))
```

[4]: '0x7559ea70'

However, look at this:

[5]:
```
a = 10
b = 10
print(hex(id(a)))
print(hex(id(b)))
```

```
0x7559eaf0
0x7559eaf0
```

The memory addresses of both **a** and **b** are the same!!

We'll revisit this in a bit to explain what is going on.

0.5 Object Mutability

Certain Python built-in object types (aka data types) are **mutable**.

That is, the internal contents (state) of the object in memory can be modified.

[1]:
```
my_list = [1, 2, 3]
print(my_list)
print(hex(id(my_list)))
```

```
[1, 2, 3]
0x1cf6ab5b208
```

[2]:
```
my_list.append(4)
print(my_list)
print(hex(id(my_list)))
```

```
[1, 2, 3, 4]
0x1cf6ab5b208
```

As you can see, the memory address of *my_list* has **not** changed.

But, the **contents** of *my_list* has changed from *[1, 2, 3]* to *[1, 2, 3, 4]*.

On the other hand, consider this:

[3]:
```
my_list_1 = [1, 2, 3]
print(my_list_1)
print(hex(id(my_list_1)))
```

```
[1, 2, 3]
0x1cf6abd55c8
```

[4]:
```
my_list_1 = my_list_1 + [4]
print(my_list_1)
print(hex(id(my_list_1)))
```

```
[1, 2, 3, 4]
0x1cf6ab56888
```

Notice here that the memory address of *my_list_1* **did** change.

This is because concatenating two lists objects *my_list_1* and *[4]* did not modify the contents of *my_list_1* - instead it created a new list object and re-assigned *my_list_1* to reference this new object.

Similarly with **dictionary** objects that are also **mutable** types.

[5]:
```
my_dict = dict(key1='value 1')
print(my_dict)
print(hex(id(my_dict)))
```

```
{'key1': 'value 1'}
0x1cf6abdcdc8
```

[6]:
```
my_dict['key1'] = 'modified value 1'
print(my_dict)
print(hex(id(my_dict)))
```

```
{'key1': 'modified value 1'}
0x1cf6abdcdc8
```

[7]:
```
my_dict['key2'] = 'value 2'
print(my_dict)
print(hex(id(my_dict)))
```

```
{'key1': 'modified value 1', 'key2': 'value 2'}
0x1cf6abdcdc8
```

Once again we see that while we are modifying the **contents** of the dictionary, the memory address of *my_dict* has not changed.

Now consider the immutable sequence type: **tuple**

The tuple is immutable, so elements cannot be added, removed or replaced.

[8]:
```
t = (1, 2, 3)
```

This tuple will **never** change at all. It has three elements, the integers 1, 2, and 3. This will remain the case as long as **t**'s reference is not changed.

But, consider the following tuple:

[9]:
```
a = [1, 2]
b = [3, 4]
t = (a, b)
```

Now, **t** is still immutable, i.e. it contains a reference to the object **a** and the object **b**. **That** will never change as long as **t**'s reference is not re-assigned.

However, the elements **a** and **b** are, themselves, mutable.

[10]:
```
a.append(3)
b.append(5)
print(t)
```

```
([1, 2, 3], [3, 4, 5])
```

Observe that the contents of **a** and **b** **did** change!

So immutability can be a little more subtle than just thinking something can never change.

The tuple **t** did **not** change - it contains two elements, that are the references **a** and **b**. And that will not change. But, because the referenced elements are mutable themselves, it appears as though the tuple has changed.

It hasn't though - that distinction is subtle but important to understand!

0.6 Function Arguments and Mutability

Consider a function that receives a *string* argument, and changes the argument in some way:

[1]:
```
def process(s):
    print('initial s # = {0}'.format(hex(id(s))))
    s = s + ' world'
```

```
    print('s after change # = {0}'.format(hex(id(s))))
```

[2]:
```
my_var = 'hello'
print('my_var # = {0}'.format(hex(id(my_var))))
```

```
my_var # = 0x1e7e96fc420
```

Note that when *s* is received, it is referencing the same object as *my_var*.

After we "modify" *s*, *s* is pointing to a new memory address:

[3]:
```
process(my_var)
```

```
initial s # = 0x1e7e96fc420
s after change # = 0x1e7e97153b0
```

And our own variable *my_var* is still pointing to the original memory address:

[4]:
```
print('my_var # = {0}'.format(hex(id(my_var))))
```

```
my_var # = 0x1e7e96fc420
```

Let's see how this works with mutable objects:

[5]:
```
def modify_list(items):
    print('initial items # = {0}'.format(hex(id(items))))
    if len(items) > 0:
        items[0] = items[0] ** 2
    items.pop()
    items.append(5)
    print('final items # = {0}'.format(hex(id(items))))
```

[6]:
```
my_list = [2, 3, 4]
print('my_list # = {0}'.format(hex(id(my_list))))
```

```
my_list # = 0x1e7e972d308
```

[7]:
```
modify_list(my_list)
```

```
initial items # = 0x1e7e972d308
final items # = 0x1e7e972d308
```

[8]:
```
print(my_list)
print('my_list # = {0}'.format(hex(id(my_list))))
```

```
[4, 3, 5]
my_list # = 0x1e7e972d308
```

As you can see, throughout all the code, the memory address referenced by *my_list* and *items* is always the **same** (shared) reference - we are simply modifying the contents (**internal state**) of the object at that memory address.

Now, even with immutable container objects we have to be careful, e.g. a tuple containing a list (the tuple is immutable, but the list element inside the tuple **is** mutable)

[9]:
```
def modify_tuple(t):
    print('initial t # = {0}'.format(hex(id(t))))
    t[0].append(100)
    print('final t # = {0}'.format(hex(id(t))))
```

[10]:
```
my_tuple = ([1, 2], 'a')
```

[11]:
```
hex(id(my_tuple))
```

[11]:
```
'0x1e7e9614288'
```

[12]:
```
modify_tuple(my_tuple)
```

```
initial t # = 0x1e7e9614288
final t # = 0x1e7e9614288
```

[13]:
```
my_tuple
```

[13]:
```
([1, 2, 100], 'a')
```

As you can see, the first element of the tuple was mutated.

0.7 Shared References and Mutability

The following sets up a shared reference between the variables my_var_1 and my_var_2

[1]:
```
my_var_1 = 'hello'
my_var_2 = my_var_1
print(my_var_1)
print(my_var_2)
```

```
hello
hello
```

[2]:
```
print(hex(id(my_var_1)))
print(hex(id(my_var_2)))
```

```
0x24c9144ca08
0x24c9144ca08
```

[3]:
```
my_var_2 = my_var_2 + ' world!'
```

[4]:
```
print(hex(id(my_var_1)))
print(hex(id(my_var_2)))
```

```
0x24c9144ca08
0x24c9144fab0
```

Be careful if the variable type is mutable!

Here we create a list (*my_list_1*) and create a variable (*my_list_2*) referencing the same list object:

[5]:
```
my_list_1 = [1, 2, 3]
my_list_2 = my_list_1
print(my_list_1)
print(my_list_2)
```

```
[1, 2, 3]
[1, 2, 3]
```

As we can see they have the same memory address (shared reference):

[6]:
```
print(hex(id(my_list_1)))
print(hex(id(my_list_2)))
```

```
0x24c9144fc48
0x24c9144fc48
```

Now we modify the list referenced by *my_list_2*:

[7]:
```
my_list_2.append(4)
```

my_list_2 has been modified:

[8]:
```
print(my_list_2)
```

```
[1, 2, 3, 4]
```

And since my_list_1 references the same list object, it has also changed:

[9]:
```
print(my_list_1)
```

```
[1, 2, 3, 4]
```

As you can see, both variables still share the same reference:

[10]:
```
print(hex(id(my_list_1)))
print(hex(id(my_list_2)))
```

```
0x24c9144fc48
0x24c9144fc48
```

0.8 ### Behind the scenes with Python's memory manager

Recall from a few lectures back:

[11]:
```
a = 10
b = 10
```

[12]:
```
print(hex(id(a)))
print(hex(id(b)))
```

```
0x7559eaf0
0x7559eaf0
```

Same memory address!!

This is safe for Python to do because integer objects are **immutable**.

So, even though a and b initially shared the same mempry address, we can never modify a's value by "modifying" b's value.

The only way to change b's value is to change it's reference, which will never affect a.

[13]:
```
b = 15
```

[14]:
```
print(hex(id(a)))
print(hex(id(b)))
```

```
0x7559eaf0
0x7559eb90
```

However, for mutable objects, Python's memory manager does not do this, since that would **not** be safe.

[15]:
```
my_list_1 = [1, 2, 3]
my_list_2 = [1, 2 , 3]
```

As you can see, although the two variables were assigned identical "contents", the memory addresses are not the same:

[16]:
```
print(hex(id(my_list_1)))
print(hex(id(my_list_2)))
```

```
0x24c9146c5c8
0x24c913c6848
```

0.9 Variable Equality

From the previous lecture we know that **a** and **b** will have a **shared** reference:

[1]:
```
a = 10
b = 10

print(hex(id(a)))
print(hex(id(b)))
```

```
0x7559eaf0
0x7559eaf0
```

When we use the **is** operator, we are comparing the memory address **references**:

[2]:
```python
print("a is b: ", a is b)
```

```
a is b:  True
```

But if we use the == operator, we are comparing the **contents**:

[3]:
```python
print("a == b:", a == b)
```

```
a == b: True
```

The following however, do not have a shared reference:

[4]:
```python
a = [1, 2, 3]
b = [1, 2, 3]

print(hex(id(a)))
print(hex(id(b)))
```

```
0x27006f17288
0x27006e968c8
```

Although they are not the same objects, they do contain the same "values":

[5]:
```python
print("a is b: ", a is b)
print("a == b", a == b)
```

```
a is b:  False
a == b True
```

Python will attempt to compare values as best as possible, for example:

[6]:
```python
a = 10
b = 10.0
```

These are **not** the same reference, since one object is an **int** and the other is a **float**

[7]:
```python
print(type(a))
print(type(b))
```

```
<class 'int'>
<class 'float'>
```

[8]:
```python
print(hex(id(a)))
print(hex(id(b)))
```

```
0x7559eaf0
0x270064b1870
```

[9]:
```python
print('a is b:', a is b)
print('a == b:', a == b)
```

```
a is b: False
a == b: True
```

So, even though a is an integer 10, and b is a float 10.0, the values will still compare as equal.

In fact, this will also have the same behavior:

[10]:
```
c = 10 + 0j
print(type(c))
```

```
<class 'complex'>
```

[11]:
```
print('a is c:', a is c)
print('a == c:', a == c)
```

```
a is c: False
a == c: True
```

0.10 ### The None Object

None is a built-in "variable" of type *NoneType*.

Basically the keyword **None** is a reference to an object instance of *NoneType*.

NoneType objects are immutable! Python's memory manager will therefore use shared references to the None object.

[12]:
```
print(None)
```

```
None
```

[13]:
```
hex(id(None))
```

[13]:
```
'0x75576bc0'
```

[14]:
```
type(None)
```

[14]:
```
NoneType
```

[15]:
```
a = None
print(type(a))
print(hex(id(a)))
```

```
<class 'NoneType'>
0x75576bc0
```

[16]:
```
a is None
```

[16]:
```
True
```

[17]:
```
a == None
```

[17]: True

[18]:
```
b = None
hex(id(b))
```

[18]: '0x75576bc0'

[19]:
```
a is b
```

[19]: True

[20]:
```
a == b
```

[20]: True

[21]:
```
l = []
```

[22]:
```
type(l)
```

[22]: list

[23]:
```
l is None
```

[23]: False

[24]:
```
l == None
```

[24]: False

0.11 Everything is an Object

[1]:
```
a = 10
```

a is an object of type **int**, i.e. **a** is an instance of the **int** class.

[2]:
```
print(type(a))
```

```
<class 'int'>
```

If **int** is a class, we should be able to declare it using standard class instatiation:

[3]:
```
b = int(10)
```

[4]:
```
print(b)
print(type(b))
```

```
10
<class 'int'>
```

We can even request the class documentation:

[5]:
```
help(int)
```

```
Help on class int in module builtins:

class int(object)
 |  int(x=0) -> integer
 |  int(x, base=10) -> integer
 |
 |  Convert a number or string to an integer, or return 0 if no arguments
 |  are given.  If x is a number, return x.__int__().  For floating point
 |  numbers, this truncates towards zero.
 |
 |  If x is not a number or if base is given, then x must be a string,
 |  bytes, or bytearray instance representing an integer literal in the
 |  given base.  The literal can be preceded by '+' or '-' and be surrounded
 |  by whitespace.  The base defaults to 10.  Valid bases are 0 and 2-36.
 |  Base 0 means to interpret the base from the string as an integer literal.
 |  >>> int('0b100', base=0)
 |  4
 |
 |  Methods defined here:
 |
 |  __abs__(self, /)
 |      abs(self)
 |
 |  __add__(self, value, /)
 |      Return self+value.
 |
 |  __and__(self, value, /)
 |      Return self&value.
 |
 |  __bool__(self, /)
 |      self != 0
 |
 |  __ceil__(…)
 |      Ceiling of an Integral returns itself.
 |
 |  __divmod__(self, value, /)
 |      Return divmod(self, value).
 |
 |  __eq__(self, value, /)
 |      Return self==value.
 |
 |  __float__(self, /)
 |      float(self)
 |
```

```
 |  __floor__(…)
 |      Flooring an Integral returns itself.
 |
 |  __floordiv__(self, value, /)
 |      Return self//value.
 |
 |  __format__(…)
 |      default object formatter
 |
 |  __ge__(self, value, /)
 |      Return self>=value.
 |
 |  __getattribute__(self, name, /)
 |      Return getattr(self, name).
 |
 |  __getnewargs__(…)
 |
 |  __gt__(self, value, /)
 |      Return self>value.
 |
 |  __hash__(self, /)
 |      Return hash(self).
 |
 |  __index__(self, /)
 |      Return self converted to an integer, if self is suitable for use as an
index into a list.
 |
 |  __int__(self, /)
 |      int(self)
 |
 |  __invert__(self, /)
 |      ~self
 |
 |  __le__(self, value, /)
 |      Return self<=value.
 |
 |  __lshift__(self, value, /)
 |      Return self<<value.
 |
 |  __lt__(self, value, /)
 |      Return self<value.
 |
 |  __mod__(self, value, /)
 |      Return self%value.
 |
 |  __mul__(self, value, /)
 |      Return self*value.
 |
```

```
|  __ne__(self, value, /)
|      Return self!=value.
|
|  __neg__(self, /)
|      -self
|
|  __new__(*args, **kwargs) from builtins.type
|      Create and return a new object.  See help(type) for accurate signature.
|
|  __or__(self, value, /)
|      Return self|value.
|
|  __pos__(self, /)
|      +self
|
|  __pow__(self, value, mod=None, /)
|      Return pow(self, value, mod).
|
|  __radd__(self, value, /)
|      Return value+self.
|
|  __rand__(self, value, /)
|      Return value&self.
|
|  __rdivmod__(self, value, /)
|      Return divmod(value, self).
|
|  __repr__(self, /)
|      Return repr(self).
|
|  __rfloordiv__(self, value, /)
|      Return value//self.
|
|  __rlshift__(self, value, /)
|      Return value<<self.
|
|  __rmod__(self, value, /)
|      Return value%self.
|
|  __rmul__(self, value, /)
|      Return value*self.
|
|  __ror__(self, value, /)
|      Return value|self.
|
|  __round__(…)
|      Rounding an Integral returns itself.
|      Rounding with an ndigits argument also returns an integer.
```

```
|
|  __rpow__(self, value, mod=None, /)
|      Return pow(value, self, mod).
|
|  __rrshift__(self, value, /)
|      Return value>>self.
|
|  __rshift__(self, value, /)
|      Return self>>value.
|
|  __rsub__(self, value, /)
|      Return value-self.
|
|  __rtruediv__(self, value, /)
|      Return value/self.
|
|  __rxor__(self, value, /)
|      Return value^self.
|
|  __sizeof__(...)
|      Returns size in memory, in bytes
|
|  __str__(self, /)
|      Return str(self).
|
|  __sub__(self, value, /)
|      Return self-value.
|
|  __truediv__(self, value, /)
|      Return self/value.
|
|  __trunc__(...)
|      Truncating an Integral returns itself.
|
|  __xor__(self, value, /)
|      Return self^value.
|
|  bit_length(...)
|      int.bit_length() -> int
|
|      Number of bits necessary to represent self in binary.
|      >>> bin(37)
|      '0b100101'
|      >>> (37).bit_length()
|      6
|
|  conjugate(...)
|      Returns self, the complex conjugate of any int.
```

```
 |
 |  from_bytes(…) from builtins.type
 |      int.from_bytes(bytes, byteorder, *, signed=False) -> int
 |
 |      Return the integer represented by the given array of bytes.
 |
 |      The bytes argument must be a bytes-like object (e.g. bytes or
bytearray).
 |
 |      The byteorder argument determines the byte order used to represent the
 |      integer.  If byteorder is 'big', the most significant byte is at the
 |      beginning of the byte array.  If byteorder is 'little', the most
 |      significant byte is at the end of the byte array.  To request the native
 |      byte order of the host system, use `sys.byteorder' as the byte order
value.
 |
 |      The signed keyword-only argument indicates whether two's complement is
 |      used to represent the integer.
 |
 |  to_bytes(…)
 |      int.to_bytes(length, byteorder, *, signed=False) -> bytes
 |
 |      Return an array of bytes representing an integer.
 |
 |      The integer is represented using length bytes.  An OverflowError is
 |      raised if the integer is not representable with the given number of
 |      bytes.
 |
 |      The byteorder argument determines the byte order used to represent the
 |      integer.  If byteorder is 'big', the most significant byte is at the
 |      beginning of the byte array.  If byteorder is 'little', the most
 |      significant byte is at the end of the byte array.  To request the native
 |      byte order of the host system, use `sys.byteorder' as the byte order
value.
 |
 |      The signed keyword-only argument determines whether two's complement is
 |      used to represent the integer.  If signed is False and a negative
integer
 |      is given, an OverflowError is raised.
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  denominator
 |      the denominator of a rational number in lowest terms
 |
 |  imag
 |      the imaginary part of a complex number
```

```
 |
 |  numerator
 |      the numerator of a rational number in lowest terms
 |
 |  real
 |      the real part of a complex number
```

As we see from the docs, we can even create an **int** using an overloaded constructor:

[6]:
```python
b = int('10', base=2)
```

[7]:
```python
print(b)
print(type(b))
```

```
2
<class 'int'>
```

0.12 ### Functions are Objects too

[8]:
```python
def square(a):
    return a ** 2
```

[9]:
```python
type(square)
```

[9]:
```
function
```

In fact, we can even assign them to a variable:

[10]:
```python
f = square
```

[11]:
```python
type(f)
```

[11]:
```
function
```

[12]:
```python
f is square
```

[12]:
```
True
```

[13]:
```python
f(2)
```

[13]:
```
4
```

[14]:
```python
type(f(2))
```

[14]:
```
int
```

A function can return a function

[15]:
```python
def cube(a):
    return a ** 3
```

[16]:
```python
def select_function(fn_id):
    if fn_id == 1:
        return square
    else:
        return cube
```

[17]:
```python
f = select_function(1)
print(hex(id(f)))
print(hex(id(square)))
print(hex(id(cube)))
print(type(f))
print('f is square: ', f is square)
print('f is cube: ', f is cube)
print(f)
print(f(2))
```

```
0x21257457b70
0x21257457b70
0x21255fab8c8
<class 'function'>
f is square:  True
f is cube:  False
<function square at 0x0000021257457B70>
4
```

[18]:
```python
f = select_function(2)
print(hex(id(f)))
print(hex(id(square)))
print(hex(id(cube)))
print(type(f))
print('f is square: ', f is square)
print('f is cube: ', f is cube)
print(f)
print(f(2))
```

```
0x21255fab8c8
0x21257457b70
0x21255fab8c8
<class 'function'>
f is square:  False
f is cube:  True
<function cube at 0x0000021255FAB8C8>
8
```

We could even call it this way:

[19]:
```python
select_function(1)(5)
```

[19]: 25

A Function can be passed as an argument to another function

(This example is pretty useless, but it illustrates the point effectively)

[20]:
```python
def exec_function(fn, n):
    return fn(n)
```

[21]:
```python
result = exec_function(cube, 2)
print(result)
```

```
8
```

We will come back to functions as arguments **many** more times throughout this course!

0.13 Python Optimizations: Interning

Earlier, we saw shared references being created automatically by Python:

[1]:
```python
a = 10
b = 10
print(id(a))
print(id(b))
```

```
1968827120
1968827120
```

Note how `a` and `b` reference the same object.

But consider the following example:

[2]:
```python
a = 500
b = 500
print(id(a))
print(id(b))
```

```
1935322088624
1935322089008
```

As you can see, the variables `a` and `b` do **not** point to the same object!

This is because Python pre-caches integer objects in the range [-5, 256]

So for example:

[3]:
```python
a = 256
b = 256
print(id(a))
print(id(b))
```

```
1968834992
1968834992
```

and

[4]:
```python
a = -5
b = -5
print(id(a))
print(id(b))
```

```
1968826640
1968826640
```

do have the same reference.

This is called **interning**: Python **interns** the integers in the range [-5, 256].

The integers in the range [-5, 256] are essentially **singleton** objects.

[5]:
```python
a = 10
b = int(10)
c = int('10')
d = int('1010', 2)
```

[6]:
```python
print(a, b, c, d)
```

```
10 10 10 10
```

[7]:
```python
a is b
```

[7]:
```
True
```

[8]:
```python
a is c
```

[8]:
```
True
```

[9]:
```python
a is d
```

[9]:
```
True
```

As you can see, all these variables were created in different ways, but since the integer object with value 10 behaves like a singleton, they all ended up pointing to the **same** object in memory.

0.14 Python Optimizations: String Interning

Python will automatically intern *certain* strings.

In particular all the identifiers (variable names, function names, class names, etc) are interned (singleton objects created).

Python will also intern string literals that look like identifiers.

For example:

[1]:
```
a = 'hello'
b = 'hello'
print(id(a))
print(id(b))
```

```
1342722069536
1342722069536
```

But not the following:

[2]:
```
a = 'hello, world!'
b = 'hello, world!'
print(id(a))
print(id(b))
```

```
1342722047024
1342722170928
```

However, because the following literals resemble identifiers, even though they are quite long, Python will still automatically intern them:

[3]:
```
a = 'hello_world'
b = 'hello_world'
print(id(a))
print(id(b))
```

```
1342722047856
1342722047856
```

And even longer:

[4]:
```
a = '_this_is_a_long_string_that_could_be_used_as_an_identifier'
b = '_this_is_a_long_string_that_could_be_used_as_an_identifier'
print(id(a))
print(id(b))
```

```
1342721886784
1342721886784
```

Even if the string starts with a digit:

[5]:
```
a = '1_hello_world'
b = '1_hello_world'
print(id(a))
print(id(b))
```

```
1342722046256
1342722046256
```

That was interned (pointer is the same), but look at this one:

[6]:
```
a = '1 hello world'
b = '1 hello world'
print(id(a))
print(id(b))
```

```
1342722046832
1342722172592
```

Interning strings (making them singleton objects) means that testing for string equality can be done faster by comparing the memory address:

[7]:
```
a = 'this_is_a_long_string'
b = 'this_is_a_long_string'
print('a==b:', a == b)
print('a is b:', a is b)
```

```
a==b: True
a is b: True
```

Note: Remember, using `is` ONLY works if the strings were interned! Here's where this technique fails:

[8]:
```
a = 'hello world'
b = 'hello world'
print('a==b:', a==b)
print('a is b:', a is b)
```

```
a==b: True
a is b: False
```

You *can* force strings to be interned (but only use it if you have a valid performance optimization need):

[9]:
```
import sys
```

[10]:
```
a = sys.intern('hello world')
b = sys.intern('hello world')
c = 'hello world'
print(id(a))
print(id(b))
print(id(c))
```

```
1342722172080
1342722172080
1342722174896
```

Notice how `a` and `b` are pointing to the same object, but `c` is **NOT**.

So, since both `a` and `b` were interned we can use `is` to test for equality of the two strings:

[11]:
```python
print('a==b:', a==b)
print('a is b:', a is b)
```

```
a==b: True
a is b: True
```

So, does interning really make a big speed difference?

Yes, but only if you are performing a *lot* of comparisons.

Let's run some quick and dirty benchmarks:

[12]:
```python
def compare_using_equals(n):
    a = 'a long string that is not interned' * 200
    b = 'a long string that is not interned' * 200
    for i in range(n):
        if a == b:
            pass
```

[13]:
```python
def compare_using_interning(n):
    a = sys.intern('a long string that is not interned' * 200)
    b = sys.intern('a long string that is not interned' * 200)
    for i in range(n):
        if a is b:
            pass
```

[14]:
```python
import time

start = time.perf_counter()
compare_using_equals(10000000)
end = time.perf_counter()

print('equality: ', end-start)
```

```
equality:  2.965451618090112
```

[15]:
```python
start = time.perf_counter()
compare_using_interning(10000000)
end = time.perf_counter()

print('identity: ', end-start)
```

```
identity:  0.28690104431129626
```

As you can see, the performance difference, especially for long strings, and for many comparisons, can be quite radical!

0.15 Python Peephole Optimizations

Peephole optimizations refer to a certain class of optimization strategies Python employs during any compilation phases.

Constant Expressions Let's see how Python reduces constant expressions for optimization purposes:

[20]:
```
def my_func():
    a = 24 * 60
    b = (1, 2) * 5
    c = 'abc' * 3
    d = 'ab' * 11
    e = 'the quick brown fox' * 10
    f = [1, 2] * 5
```

[21]:
```
my_func.__code__.co_consts
```

[21]:
```
(None,
 24,
 60,
 1,
 2,
 5,
 'abc',
 3,
 'ab',
 11,
 'the quick brown fox',
 10,
 1440,
 (1, 2),
 (1, 2, 1, 2, 1, 2, 1, 2, 1, 2),
 'abcabcabc')
```

As you can see in the example above, `24 * 60` was pre-calculated and cached as a constant (`1440`).

Similarly, `(1, 2) * 5` was cached as `(1, 2, 1, 2, 1, 2, 1, 2, 1, 2)` and `'abc' * 3` was cached as `abcabcabc`.

On the other hand, note how `'the quick brown fox' * 10` was **not** pre-calculated (too long).

Similarly `[1, 2] * 5` was not pre-calculated either since a list is *mutable*, and hence not a *constant*.

Membership Tests In membership testing, optimizations are applied as can be seen below:

[69]:
```
def my_func():
    if e in [1, 2, 3]:
        pass
```

[70]:
```python
my_func.__code__.co_consts
```

[70]:
```
(None, 1, 2, 3, (1, 2, 3))
```

As you can see, the mutable list `[1, 2, 3]` was converted to an immutable tuple.

It is OK to do this here, since we are testing membership of the list **at that point in time**, hence it is safe to convert it to a tuple, which is more efficient than testing membership of a list.

In the same way, set membership will be converted to frozen set membership:

[22]:
```python
def my_func():
    if e in {1, 2, 3}:
        pass
```

[23]:
```python
my_func.__code__.co_consts
```

[23]:
```
(None, 1, 2, 3, frozenset({1, 2, 3}))
```

In general, when you are writing your code, if you can use **set** membership testing, prefer that over a list or tuple - it is quite a bit more efficient.

Let's do a small quick (and dirty) benchmark of this:

[5]:
```python
import string
import time

char_list = list(string.ascii_letters)
char_tuple = tuple(string.ascii_letters)
char_set = set(string.ascii_letters)

print(char_list)
print()
print(char_tuple)
print()
print(char_set)
```

```
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p',
'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z', 'A', 'B', 'C', 'D', 'E', 'F',
'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V',
'W', 'X', 'Y', 'Z']

('a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p',
'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z', 'A', 'B', 'C', 'D', 'E', 'F',
'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V',
'W', 'X', 'Y', 'Z')

{'l', 'p', 'x', 'R', 'j', 'S', 's', 'T', 'W', 'Y', 'Z', 'P', 'g', 'O', 'b', 'u',
'H', 'G', 'v', 'e', 'M', 'n', 'w', 't', 'Q', 'E', 'N', 'X', 'C', 'i', 'A', 'B',
```

'F', 'V', 'a', 'm', 'r', 'f', 'h', 'U', 'D', 'c', 'y', 'z', 'J', 'd', 'o', 'I',
'L', 'K', 'k', 'q'}

[6]:
```
def membership_test(n, container):
    for i in range(n):
        if 'p' in container:
            pass
```

[7]:
```
start = time.perf_counter()
membership_test(10000000, char_list)
end = time.perf_counter()
print('list membership: ', end-start)
```

```
list membership:  2.6035404184015434
```

[8]:
```
start = time.perf_counter()
membership_test(10000000, char_tuple)
end = time.perf_counter()
print('tuple membership: ', end-start)
```

```
tuple membership:  2.602491734651276
```

[9]:
```
start = time.perf_counter()
membership_test(10000000, char_set)
end = time.perf_counter()
print('set membership: ', end-start)
```

```
set membership:  0.3743007599607324
```

As you can see, set membership tests run quite a bit faster - which is not surprising since they are basically dictionary-like objects, so hash maps are used for looking up an item to determine membership.