From Perl to Python

Author: Fernando Pineda Created: Oct. 20, 2016 Update: Oct. 1, 2018 version 0.2

Preliminaries

We will learn Python3. Python3 is NOT backward compatible with Python2.

You will need to follow along in the python3 interpreter on the cluster. To do this, first log into the cluster and load the Python 3.6.3 module. Then check your python version number:

```
module load python/3.6.3
python3 --version
```

You can start the python3 interpreter by entering `python3` at the command line. Once in python, you can quit by entering `exit()`.

1. Everything is an object!

Yes. Literally everything is an object. More precisely, everything is a pointer to an object. To really appreciate these statements we will analyze a single, absurdly trivial, Python statement:

```
>>> "foo"
foo
```

That's it. What happened? This is what happened: 1) Python created an *object* of type `string` in the memory 2) Python saved the *character string* "foo" in this *string object*, 3) Python echoed the *character string* to the console and finally, 4) Python marked the *string object* for deletion.

Now let's use three functions to explore and understand all this in more detail:

1. `id()` Every object has a unique identifier. We can get at the identifier with the `id()` function.
2. `type()` Every object has a type. We can view the type with the `type()` function.
3. `dir()` Every object has a list of names defined in it's scope. We can view the names with the `dir()` function.

We'll get a more precise notion of *names* and *scope* later on. For now let's just use these functions to get a deeper understanding of Python. Consider the following:

```
>>> "foo"  # create a string object and print it
foo
>>> id("foo")     # create a string object and print its unique id
4449499456
>>> id("foo")     # create another string object and print its unique id
4449499416
>>> id("foo")     # create yet another string object and print its unique id
4449499576
```

Although it is not evident here, it's important to understand that every time we enter "foo", Python creates a brand new string object to hold the string and that after each statement is executed, Python marks the object for destruction. In Python objects exist transiently (to save memory) unless we assign them a name. More about this later.

All objects have methods and attributes that have assigned names. Let's use the `dir()` function to view the names of the methods and attributes in our string objects:

```
>>> dir("foo") #  and with all these names
['__add__', '__class__', '__contains__', '__delattr__', '__doc__', '__eq__', '__format__',
'__ge__', '__getattribute__', '__getitem__', '__getnewargs__', '__getslice__', '__gt__',
'__hash__', '__init__', '__le__', '__len__', '__lt__', '__mod__', '__mul__', '__ne__',
'__new__', '__reduce__', '__reduce_ex__', '__repr__', '__rmod__', '__rmul__', '__setattr__',
'__sizeof__', '__str__', '__subclasshook__', '_formatter_field_name_split',
'_formatter_parser', 'capitalize', 'center', 'count', 'decode', 'encode', 'endswith',
'expandtabs', 'find', 'format', 'index', 'isalnum', 'isalpha', 'isdigit', 'islower',
'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip', 'partition', 'replace',
'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines',
'startswith', 'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']
>>>
```

We will understand the naming conventions for methods later on. For now just look at the methods and attributes without underscores in their names. These are the ones that are available for you to use in your programs.

Python is strongly typed. This is like C and Java and unlike Perl. So when you create an object, it has a type and can only hold data of that type. For now we will only consider the built-in types, e.g. int, long, float, complex, string, list, tuple, set, frozenset, dict, etc.

Now let's try out a couple of the string object methods. The syntax for invoking a method is as follows: <object_name>.<method_name>(). This is reminiscent of how we invoke methods in Perl (<object_name>-><method_name>().). Accordingly, let's try the `upper` method:

```
>>> "foo".upper() # convert to upper case
FOO
```

In this statement Python creates two string objects. First a string object to hold the string "foo". Then the `upper` method returns a new object with the "FOO" string. Finally, after the statement is executed, Python marks the both string objects (the one that holds "foo" and the one that holds "FOO") for deletion. The objects are not actually deleted until Python needs the memory. This can happen at any time. Now let's try a method that takes an argument:

```
>>> "foo".__add__("bar") # __add__ concatenates two str objects
foobar
>>> id( "foo".__add__("bar") ) # the __add__ method returns another object
4449552640
```

In the first statement Python creates two string objects for the "foo" and "bar" strings. Then the `__add__` method of the "foo" object is invoked with the "bar" object as its argument. The method returns a new object with the concatenated string. All the same things happen in the third statement except that the returned object is passed to the id() function. All three objects are marked for destruction after each statement.

2. Names and assignment

In C, Perl, Java and many other languages, we can view variables as names for containers where values are stored. We replace values in the containers with new values via the assignment operator. For example, in Perl:

```
$a = 5;  # the value 5 is saved in the variable $a
$a = 6;  # the value 6 replaces the value 5 in the variable $a
$b = $a; # the contents of the variable $a (6) replaces the value in $b
```

Python is different. Instead of "variables" Python has *names* and names are just references (pointers) to objects. Objects do not persist unless they have a name attached to them. Another way to say this is that objects do not persist unless they have a name pointing to them. Names are attached to objects via the assignment operator. An assignment statement looks like this:

name = *expression*

Python evaluates the expression on the right hand side and creates an object of the appropriate type to hold the result. If there is no assignment, it ends there. On the other hand, if there is an assignment, it then associates the name on the LHS to the object. Note that a name is not part of an object. Instead Python maintains names and their references to objects in tables known as *namespaces*.

To begin appreciating the meaning and implication of these words consider the following:

```
>>> a="foo"
>>> id(a)
4449501016
>>> id(a)
4449501016
```

In the first statement Python creates a string object for "foo" and then attaches the name 'a' to the object. Since the string object that holds "foo" is now associated with the name 'a', it will persist. Accordingly, when we invoke `id()` the name 'a', `id()` now always returns the unique id of the persistent string object.

Up to this point the distinction between names and variables might seem a little pointless, but now consider the following:

```
>>> x = [1,0]
>>> y = x
>>> print y
[1, 0]
>>> x[1] = -1
>>> print x
[1, -1]
>>> print y
[1, -1]
```

If you are still thinking of variables as containers instead of names, then the last line will startle you since it appears that changing one variable changed another. Perhaps the following two lines will shed some light:

```
>>> id(x)
4449382200
>>> id(y)
4449382200
```

These two lines reveal that the two names 'x' and 'y' are both associated with the same object. This was accomplished by the statement `y = x`.

It is now clear what Python is doing. The statement `x[1] = -1` changed the second element of the list whose unique id is 4449382200. We referred to that list with the name 'x' *and* with the name 'y'.

If you are a C or perl programmer you will recognize this as the behavior of pointers (in C) or references (in Perl). The object id plays the role of the Perl reference. In perl this would look like this:

```
$x = [1,0];     # create a reference to the list (1,0) and save it in $x
$y = $x;        # copy the reference to $y
print @$y;      # dereference the array reference and print it
$x->[1] = -1;   # save -1 in the second element of the list
print @$y;      # dereference the array reference and print it
```

To summarize: In effect, names in Python are references (pointers) to objects. There is no problem with having multiple pointers pointing to the same object. If you internalize that one thought, you will be happy as a clam when you write your code and you will avoid much of the confusion that arises when debugging your Python code.

3. Functions

The sub statement in Perl

In Perl (also in C and Java), there are two phases in processing a script or program. The first phase involves compilation of the language statements. The compiler for the language converts statements to either machine code or byte code and creates structures in the memory representing global variables and functions. The byte code is subsequent execution by the (virtual) machine. *compile time* refers to the compilation phase while *run time* refers to the execution phase.

In Perl, functions are created by declaring them at compile-time via the `sub` statement. So `sub` is not an executable statement. It is only understood by the compiler and is used by the compiler to create the executable code for the subroutine. A function is subsequently invoked by its name together with passed arguments, e.g.

```
# declaring the function adder
sub adder {
  my ($x,$y) = @_;
  return $x+$y;
}

# invoking the adder function
my $z = adder(4,5)
```

The def statement in Python

In Python, functions are not declared at compile-time. Instead, functions, like everything else in Python are objects. The `def` statement is executed at run time and creates an object of type `function` and assigns a name to the object, e.g.

```
# creating the object adder
def adder(x,y) :
  return x+y

# invoking adder
z = adder(4,5)
```

We can examine the adder function in the usual way with the `id()`, `type()` and `dir()` functions.

```
>>> type(adder)
```

```
<type 'function'>
>>> id(adder)
4466549416
>>> dir(adder)
['__call__', '__class__', '__closure__', '__code__', '__defaults__', '__delattr__',
'__dict__', '__doc__', '__format__', '__get__', '__getattribute__', '__globals__',
'__hash__', '__init__', '__module__', '__name__', '__new__', '__reduce__', '__reduce_ex__',
'__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', 'func_closure',
'func_code', 'func_defaults', 'func_dict', 'func_doc', 'func_globals', 'func_name']
```

Unlike Perl function names, Python function names DO NOT have global scope. We will return to the scoping rules in a moment.

The lamdba statement (anonymous functions)

Python doesn't actually have general anonymous functions like in Perl. However, it does allow a more limited kind of anonymous expressions called a 'lambda' or a 'lambda expression'. lambdas are objects of type `function`, but multiple statements are not allowed in lambdas. They must have one and only one statement and no calls to `print` and no `return` statements. This restriction on lambda purposely restricts it's utility. Why? Ask Guido. It is what it is.

```
# here we create a function object with a lambda expression and assign the name 'f' to it.
f = lambda x,y : x+y

# this is the same as
def f(x,y):
  return x+y
```

Passing arguments to functions

Passing arguments in Python is essentially pass-by-reference. When we put an unnamed object in the argument list, e.g. "foo" or a lambda, the id of the object is passed to the function.

Inside the function, Python binds the object to the corresponding name in the function prototype. Similarly, if we put a name in the argument list, Python takes the object reference associated with the name and passes the reference to the function, where the references is bound to the corresponding argument in the argument list. For example consider the following code:

```
# create the function object and assign it to the name 'f'
def f (x):
    print id(x)

a = 'foo'
print id(a)  # prints 140442759877704 from outside the function
f(a)         # prints 140442759877704 from inside the function
```

The name 'a' is bound to the object containing the number '5'. The name is passed to the function 'f'. This results in the object holding 5 being assigned to the name 'x' inside the function. Thus the result is that the print statement outside the function and the print statement inside the function, both produce the same object id because they both point to the same object.

4. Scoping rules

Perl and Python have very different scoping rules. We'll start with a review of Perl and then we'll flush these ideas from

our brains.

Quick review of Perl

Perl scalars, arrays, hashes are either globals or lexicals. Globals exist in the namespace where they were declared. In the main program, this namespace is `main`. In modules, it is the namespace declared in the `Package` statement. Globals created in a namespace are generally visible throughout the entire scope of the namespace, e.g. in a code block.

Perl has nested scoping rules for lexical variables. In other words, Lexical variables declared in a code block are visible within that code block and in all inner code blocks. However, they do not 'leak' out to outer scopes.

If, by chance, a lexical variable is declared with the same name as a variable (lexical or global) in an outer scope, then the new lexical variable is visible (and the outer variable is invisible) from the point it is declared until execution exists the local scope:

```
my $x = 0;    # lexical variable declared and assigned in the outer scope

{             # new code block creates an inner scope
  print $x;   # prints '0'
  my $x = 1;  # new lexical visible from here until execution exits inner scope
  print $x;   # prints '1'
}             # end of the code block and the inner scope

print $x      # prints '0' since we're back to the outer scope
```

Python's scoping rules

The notions of "scope" and "namespace" are related. A namespace is a table that contains a mapping between a name and an object. It is a many to one map, i.e. a name can map to just one object, but many names can map to a single a single object.

When we ask about the scope of a variable, we are asking, which namespace (table), python will use to find an object.

The following article has an excellent presentation of Python's scoping rules:

https://sebastianraschka.com/Articles/2014_python_scope_and_namespaces.html

Let's first consider local, global and built-in names.

By default, all variables are global unless they are declared inside a function. In other words, there is the namespace associated with the file level and then each function has it's own namespace.

Outside of functions Python will search for a name in the global namespace. Inside a function, Python will first search the function's namespace, and if the name is not there, it will search the global namespace. Finally, if the name is not in the global namespace, Python will search for *built-in* names. This order of search goes by the name of the **LGB** rule. The order of the search determines the scope of a variable.

Let's explore Python scope with a few examples. Consider the following Perl code:

```
@items = (1,2,3,4,5);

$i = 0;
for my $i (@items) {
  ;
```

```
}
print $i, "\n";
```

The for loop does nothing but step through the elements of the items list. It prints '0'. As would be expected since the scope of the lexical $i in the for statement is limited to the iterated code block. Now consider superficially similar code in Python:

```
items = [1,2,3,4,5]
i=0
for i in items:
    pass
print(i)
```

In this example the `pass` statement is the empty statement (equivalent to `;` in Perl). There are no functions in this snippet. So there is only one namespace -- the global namespace. Consequently, this prints `5` since 5 is the last value that `i` takes.

Next, recall that functions have their own private namespace in addition to the global namespace. AND Python searches the function namespace before it searches the global namespace when it searches for a name. Consider the following Python snippet:

```
x = 0
def foo(y):
  print x,y

print(foo(1))
```

This prints '0 1'. The value of `x` comes from the outer scope while the value of `y` is passed as a function argument. Now consider this:

```
x = 0
def foo(y):
  x = -1
  print x,y

print(foo(1))
```

Not surprisingly this produces '-1,1'. But what about this:

```
x = 0
def foo(y):
  print x,y
  x = -1

print(foo(1))
```

From our experience with Perl, we might expect exactly the same result as before, but instead we get an error message:

```
  Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
    File "<stdin>", line 2, in foo
  UnboundLocalError: local variable 'x' referenced before assignment
```

Here is why this happens: The `def` statement causes Python to create a function object. However, Python does not execute the code in the function object until the function is actually invoked (in the print statement). When Python creates the function object, the local name `x` gets put into the function's namespace.

When the function code is actually executed, Python finds `x` in the function's own namespace, so it doesn't look in the global namespace. Unfortunately, the object that the local `x` points to has not been initialized to a value yet. So attempting to print an uninitialized value results in an error. Python complains that we have referenced the object before assigning it a value.

Scope = The hierarchy of namespace search

The LGB rule covers most cases of interest for casual Python programmers. However, the secret to really understanding scope, is to understand: 1) *when* Python saves names, 2) *where* Python saves names and 3) *the order* in which it searches tables for names.

First recall that When a `def` statement is executed, it creates a function object and then associates a name with that object. Up to this point, we have only looked at examples where Python saves function names in the global namespace table.

Function object are fairly complicated objects. For our purposes we only care about two things that every function object contains:

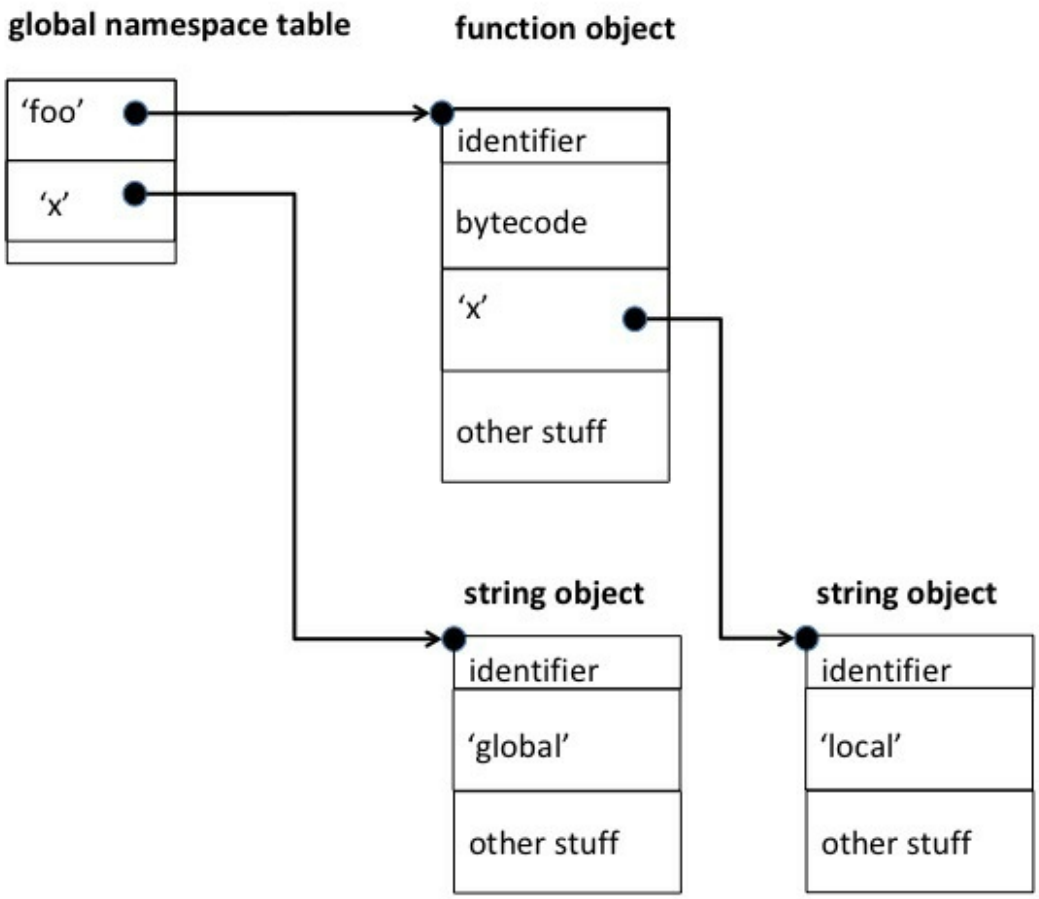
The Bytecode When encountering a `def` statement, Python compiles the statements into *bytecode* and saves the bytecode in the object it constructs. **Note: The bytecode is not executed when a function object is constructed. The bytecode is only executed when the function is actually invoked**. If you really care, you can view a function's bytecode with the Python disassembler function `dis.dis()`.

The local table of names Every function object has its own private table of names. This table is used to save local names and pointers to the objects that they refer to. Names in the local table are saved and read as the function's bytecode is executed.

First let's review how Python compiles and executes a simple script into objects.

```
def foo():
        x = 'local'
        print("x = ",x)


x = "global"
foo()
print("x = ",x)
```

The figure below provides a notional picture of the namespaces and objects that are created.

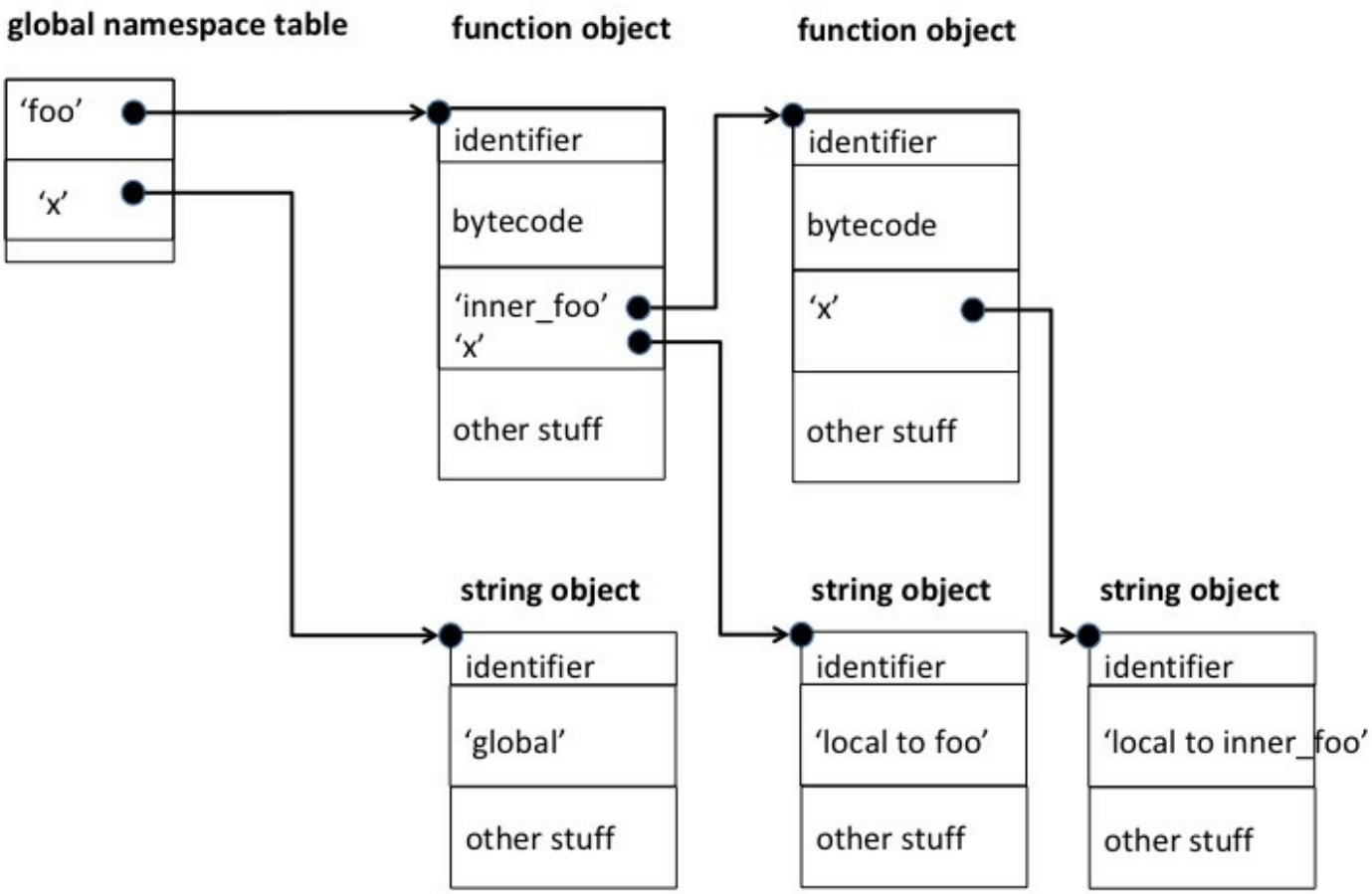


Note that the local namespace table can't get built untilthe function's statements (bytecode) is actually executed. When these statements are executed, python will do name resolution in LEGB order (L= local, E= Enclosing, G=Global, B=built-in).

What do we mean by 'Enclosing'? Consider a function defined within another function.

```
def foo():
        x = "local to foo"

        def inner_foo():
                # x = "local to inner_foo"
                print("x = ", x)

        print("x = ",x)
        inner_foo()
        print("x = ",x)

x = "global"

foo()
print("x = ",x)
```

Here are the tables and objects that are created


global namespace table
function object
function object
'foo'
'x'
identifier
bytecode
'inner_foo'
'x'
other stuff
identifier
bytecode
'x'
other stuff
string object
string object
string object
identifier
'global'
other stuff
identifier
'local to foo'
other stuff
identifier
'local to inner_foo'
other stuff