From Perl to Python

Fernando Pineda 140.636 Oct. 11, 2017 version 0.1

3. Strong/Weak Dynamic typing

The notion of strong and weak typing is a little murky, but I will provide some examples that provide insight most.

Strong vs Weak typing To be *strongly typed* means that the type of data that a variable can hold is fixed and cannot be changed.

To be *weakly typed* means that a variable can hold different types of data, e.g. at one point in the code it might hold an `int` at another point in the code it might hold a `float`.

Static vs Dynamic typing

To be *statically typed* means that the type of a variable is determined **at compile time** . Type checking is done by the compiler prior to execution of any code.

To be *dynamically typed* means that the type of variable is determined at run-time. Type checking (if there is any) is done when the statement is executed.

C is a *strongly* and *statically* language.

```
float x;
int y;

# you can define your own types with the typedef statement
typedef mytype int;
mytype z;

typedef struct Books {
  int book_id;
  char book_name[256]
}
```

Perl is a *weakly* and *dynamically* typed language. The type of a variable is set when it's value is first set Type checking is performed at run-time. Here is an example from stackoverflow.com that shows how the type of a variable is set to 'integer' at run-time (hence dynamically typed). `$value` is subsequently cast back and forth between string and a numeric types (hence weakly typed).

```

  $value = 42;

  #concat 0 onto the end.
  $value .= 0;
  print $value,"\n";
  #numeric add
  $value += 1;
  print $value,"\n";
  #division - to create a float.
  $value = $value / 4;
  print $value,"\n";
  #string replacement on a float.
  $value =~ s/\.25//g;
  print $value,"\n";

```

Python is also a *weakly* and *dynamically* typed language. In the example below the name `x` is first bound to a numeric object (with value 1). We verify that it's type is `int` with the `type()` function. Next we bind `x` to an object of type `float`. We then verify that it's type has changed. In the above description I have been careful to distinguish between the name that is bound to the object and the type of an object. This is to reinformce that the type of an objects does not change, but names can be rebound to objects with different types. Thus Python behaves like a weakly typed language.

```
>>> x = 1
>>> type(x)
<type 'int'>
>>> x = 1.1
>>> type(x)
```

```
<type 'float'>
```

4. Functions and scope

Declaring functions in Perl with the sub statement

In Perl (also in C and Java), functions are declared at compile-time via the sub statement. 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;
}

# we invoke the adder function like this
my $z = adder(4,5)
```

Declaring functions in Python with the def statement

Note: Python is not a compiled language. It is an interpreted language, which means that the Python interpreter slogs through the statements in your program as it encounters them. In particular, this means that if you try to invoke a function that has not been declared prior to the invocation, Python will throw an error.

This in contrast to C, Perl and Java which are compiled languages. In these languages, the compiler parses your entire program file and converts the declarations and statements into binary code or byte code, which is then executed by either a real (binary code) or virtual (byte code) machine. In modern compiled languages the compiler makes multiple passes through the entire program file prior to emitting any binary or byte code. Thus modern compilers are not bothered if function declarations are defined downstream of their invocation statement. We distinguish between compile-time and run-time. In compiled languages function declarations are executed at compile time, but are invoked at runtime. In interpreted languages function declarations must be executed at run-time prior to their invocation at run time.

The `def` statement 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.

Passing arguments to functions

Passing arguments in Python is pass-by-reference. When we put an unnamed object in the argument list, e.g. "foo", the id of the object is passed to the function. Inside the function, Python binds the name in the prototype to the passed object. 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 e.g., 140442759877704 from outside the function
f(a)         # prints 140442759877704 again 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.

The lambda statement (anonymous functions)

Python doesn't actually have general anonymous functions like in Perl. However, it does allow a more limited kind of anonymous expression called a 'lambda' or a 'lambda expression'. lambdas are objects of type `function`. Unfortunately multiple statements are not allowed in lambdas. They must have one and only one statement and no calls to `print` or `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
```

4. Scoping rules

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

Perl scoping rules allow nesting of 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, lexical variables declared in code blocks are not visible outside the code block. 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 exits 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 functions and scope

Python has local variables and global variables. But **Python does not have the notion of nested scope**. Code blocks do not create inner scopes. The only way to introduce a new inner scope is to create a new function object. And the only way to nest a scope is to create a function within a function. By default any names that are assigned inside a function are local to the function and do not leak out to the enclosing scope. Moreover, even if you create functions within functions, the names in the outer scopes will only be visible one level deeper. So even functions within functions within functions can cause you grief.

Now 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). This prints `5` since the iterated code block does not create an inner scope and the last value that `i` takes is `5`.

Functions have their own inner scope and names in the outer scope are visible inside the function (but only one level deep) e.g.

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

# now execute foo
x = 0
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:

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

# now execute foo
x = 0
foo(1)
```

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

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

# now execute foo()
x  = 0
foo(1)
```

We might expect '0,1'the same result. 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
```

What happen? Parsing and creating a function object is as close as Python gets to compiling. When Python encounters a function, parses it and puts the code into an object of type 'function'.

What happened is that when Python parsed the function, it parsed the entire function. When it did this it encountered the statement `x = -1`. This caused it to bind the name 'x' to the object containing -1. The key point in the previous statement is that inside the **entire inner scope of foo()** 'x' is a local variable. In Perl, the scope of the lexical 'x' would have neen from the statement where it was introduced (x=-1) to the end of the scope. In Perl, above the print statement 'x' referred to the outer 'x'.

Now back to Python. When Python tried to execute the `print x,y` statement, it encountered a local variable, x. But the local x did not yet have a value. Hence the error.

We can force python to use the outer `x` instead of the inner `x` by declaring `x` to be `global`.

```
def foo(y):
  global x
  print x,y
  x = -1
# now invoke foo()
x=0
foo(1)
(0, 1)
```

Now all is right with the world again. Except that x is not a global in the sense of Perl or C. All we have done with the global statement is to tell Python to use the x from the outer scope. NOT a global x in the sense of Perl or C.

~~~
### Name resolution: The LGB rule

The LGB rule refers to the order of the namespaces that Python searches when it is trying
to resolve names. LGB refers to the three different scopes that are searched: L= local,
G=global and B=built-in. Consider the following example:



```Python
x = 1               # x and foo are assigned in the global scope

def foo(y):
    z = x+y         # x is not assigned, so this x is from the enclosing scope
    return z        # y is assigned when the function is invoked, so it is local
                    # z is assigned in the z = x+y statement, so it is local
print(foo(2))
~~~

This prints '3'. Finally, we can let a name have global scope by declaring the name as a global. The example below is exactly the same as the example above, except we have declared the name 'z' to be global so that it is globally visible

```
x = 1

def foo(y):
    global z  # this allows z to be visible in the enclosing namespace
    z = x+y
    return z

print(foo(2))
print z
```

This prints '3' twice. Once from the return value and once from the global z. As always, it is usually a bad idea to use globals.