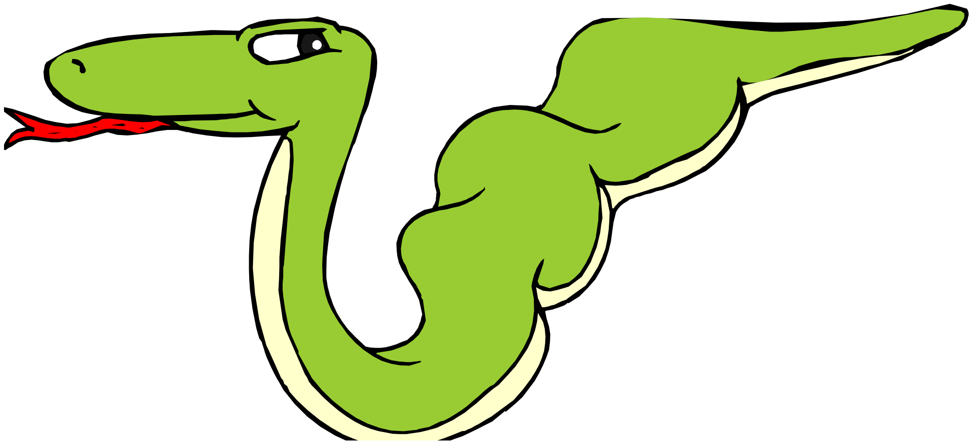
Assignment and Containers



Multiple Assignment with Sequences

- We've seen multiple assignment before:

```
>>> x, y = 2, 3
```

- But you can also do it with sequences.
  - The "shape" has to match.

```
>>> (x, y, (w, z)) = (2, 3, (4, 5))
>>> [x, y] = [4, 5]
```

Empty Containers 1

- **Assignment creates a name, if it didn't exist already.**

  `x = 3` **Creates name x of type integer.**

- **Assignment is also what creates named references to containers.**

```
>>> d = {'a':3, 'b':4}
```

- **We can also create empty containers:**

```
>>> li = []
>>> tu = ()
>>> di = {}
```

Note: an empty container is *logically* equivalent to False. (Just like None.)

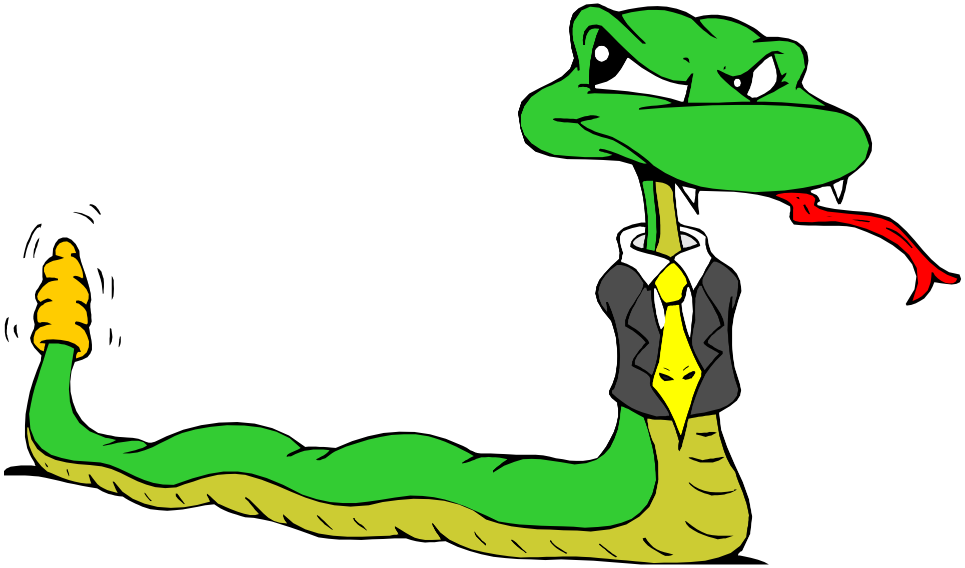
- **These three are empty, but of different *types***

Empty Containers 2

- **Why create a named reference to empty container?**
  - To initialize an empty list, for example, before using append.
  - This would cause an unknown name error a named reference to the right data type wasn't created first

```
>>> g.append(3)
Python complains here about the unknown name 'g'!
>>> g = []
>>> g.append(3)
>>> g
[3]
```

Importing and Modules



Importing and Modules

- **Use classes & functions defined in another file.**
- **A Python module is a file with the same name (plus the *.py* extension)**
- **Like Java *import*, a little bit like C++ *include*.**
- **Three formats of the command:**

```
import somefile
from somefile import *
from somefile import className
```

What's the difference?

What it is that is imported from the file and how we refer to the items after import.

import ...

```
import somefile
```

- *Everything* in somefile.py gets imported.
- To refer to something in the file, append the text ″somefile.″ to the front of its name:

```
somefile.className.method("abc")
somefile.myFunction(34)
```

*from … import **

```
from somefile import *
```

- *Everything* in somefile.py gets imported
- To refer to anything in the module, just use its name. Everything in the module is now in the current namespace.
- *Caveat!* Using this import command can easily overwrite the definition of an existing function or variable!

```
className.method("abc")
myFunction(34)
```

from … import …

```
from somefile import className
```

- **Only the item *className* in somefile.py gets imported.**
- **After importing *className*, you can just use it without a module prefix. It's brought into the current namespace.**
- ***Caveat*! This will overwrite the definition of this particular name if it is already defined in the current namespace!**

```
className.method("abc")      ← This was imported by the command.
myFunction(34)               ← This one wasn't!
```

Commonly Used Modules

- Some useful modules to import, included with Python:

- Module: sys - Lots of handy stuff.
  - Maxint
- Module: os - OS specific code.
- Module: os.path - Directory processing.

More Commonly Used Modules

- **Module: math** **- Mathematical code.**
  - Exponents
  - sqrt
- **Module: Random** **- Random number code.**
  - Randrange
  - Uniform
  - Choice
  - Shuffle

Defining your own modules

- You can save your own code files (modules) and import them into Python.

Directories for module files

Where does Python look for module files?

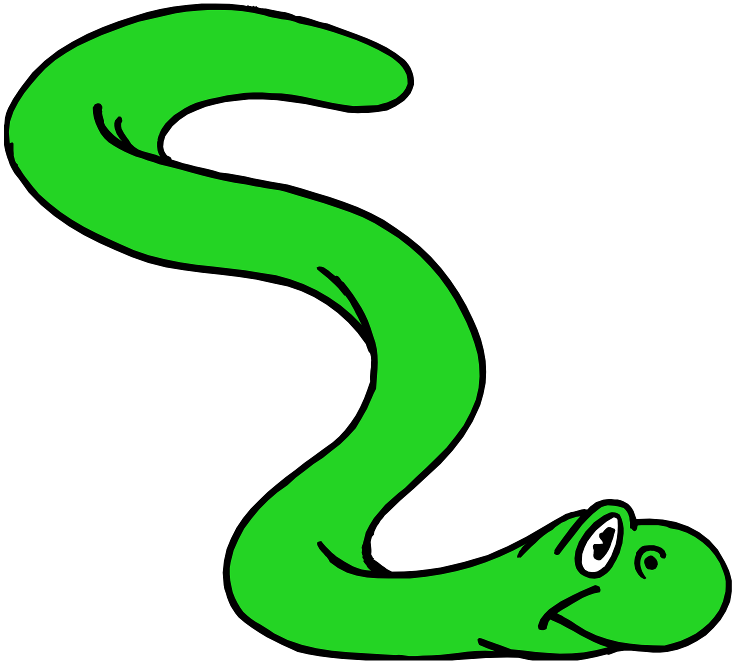
- **The list of directories in which Python will look for the files to be imported: sys.path**

  (Variable named ′path′ stored inside the ′sys′ module.)

- **To add a directory of your own to this list, append it to this list via a statement in your script.**

  ```
  sys.path.append('/my/new/path')
  ```

Object Oriented Programming in Python: Defining Classes



It’s all objects…

- **Everything in Python is really an object.**
  - We’ve seen hints of this already…

    ```
    ”hello”.upper()
    list3.append(’a’)
    dict2.keys()
    ```

  - These look like Java or C++ method calls.
  - New object classes can easily be defined in addition to these built-in data-types.
- **In fact, programming in Python is typically done in an object-oriented fashion.**

Defining a Class

- **A *class* is a special data type which defines how to build a certain kind of object.**
  - The *class* also stores some data items that are shared by all the instances of this class.
  - *Instances* are objects that are created which follow the definition given inside the class.
- **Python doesn’t use separate class interface definitions as in some languages.**
  - You just define the class and then use it.

Methods in Classes

- **Define a *method* in a *class* by including function definitions within the scope of the class block.**
  - There must be a special first argument `self` in *all* method definitions which gets bound to the calling instance
  - There is usually a special method called `__init__` in most classes
  - We'll talk about both later…

A simple class definition: *student*

```
class student:
  """A class representing a student."""
  def __init__(self, n, a):
      self.full_name = n
      self.age = a
  def get_age(self):
      return self.age
```

Creating and Deleting Instances

Instantiating Objects

- There is no ″new″ keyword (i.e. Python is not the same syntactically as Java).
- Merely use the class name with () notation and assign the result to a variable.
- `__init__` serves as a constructor for the class. Usually does some initialization work (of course).
- The arguments passed to the class name are given to its `__init__()` method.
  - So, the __init__ method for student is passed ″Bob″ and 21 here and the new class instance is bound to b:

```
b = student(″Bob″, 21)
```

Constructor: __init__

- **An `__init__` method can take any number of arguments.**
  - Like other functions or methods, the arguments can be defined with default values, making them optional to the caller.

- **However, the first argument `self` in the definition of __init__ is special…**

self

- **The first argument of every method is a reference to the current instance of the class.**
  - By convention, we name this argument ***`self`***.
  - We could give it a different name, but we'd risk writing unreadable Python code…

- **In `__init__`, *`self`* refers to the object currently being created; so, in other class methods, it refers to the instance whose method was called.**
  - Similar to the keyword *this* in Java or C++.
  - But Python uses *self* more often than Java uses *this*.

self

- Although you must specify `self` explicitly when ***defining*** the method, **you don't include it** when ***calling*** the method.
- Python passes it for you automatically.

Defining a method:

(this code inside a class definition.)

```
def set_age(self, num):
   self.age = num
```

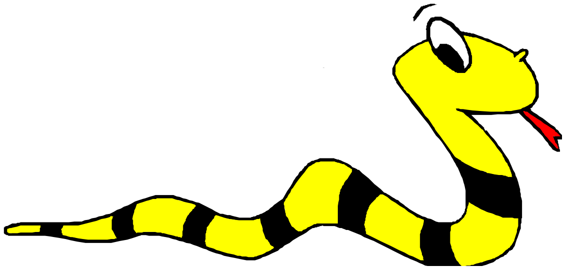
Calling a method:

```
>>> x.set_age(23)
```

Deleting instances: No Need to ″free″

- **When you are done with an object, you don′t have to delete or free it explicitly.**
  - Python has automatic garbage collection.
  - Python will automatically detect when all of the references to a piece of memory have gone out of scope. Automatically frees that memory.
  - Generally works well, few memory leaks.
  - There′s also no ″destructor″ method for classes.

Access to Attributes and Methods



Definition of student

```
class student:
  """A class representing a student."""
  def __init__(self, n, a):
      self.full_name = n
      self.age = a
  def get_age(self):
      return self.age
```

Traditional Syntax for Access

```
>>> f = student ("Bob Smith", 23)

>>> f.full_name    # Access an attribute.
"Bob Smith"

>>> f.get_age()     # Access a method.
23
```

Accessing unknown members

- **Problem: Occasionally the name of an attribute or method of a class is only given at run time…**

- **Solution: `getattr(object_instance, string)`**
  - **`string`** is a string which contains the name of an attribute or method of a class
  - **`getattr(object_instance, string)`** returns a reference to that attribute or method

- **Only need this when writing <u>very</u> extensible code**

getattr(object_instance, string)

```
>>> f = student("Bob Smith", 23)

>>> getattr(f, "full_name")
"Bob Smith"

>>> getattr(f, "get_age")
 <method get_age of class studentClass at 010B3C2>

>>> getattr(f, "get_age")()    # We can call this.
23

>>> getattr(f, "get_birthday")
      # Raises AttributeError - No method exists.
```

hasattr(object_instance,string)

```
>>> f = student("Bob Smith", 23)

>>> hasattr(f, "full_name")
True

>>> hasattr(f, "get_age")
True

>>> hasattr(f, "get_birthday")
False
```