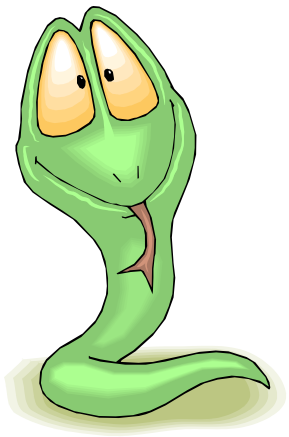
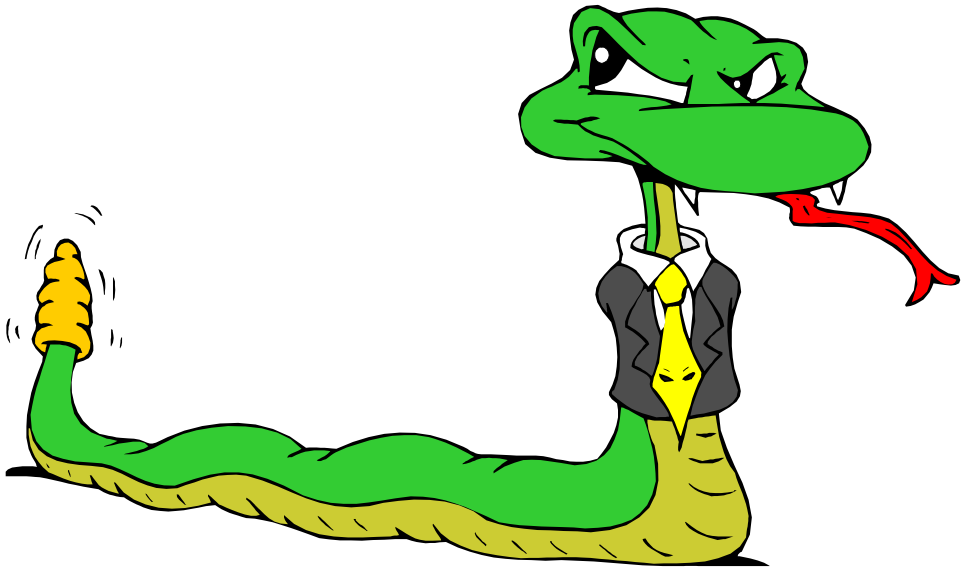
Intro to Python

- Modules
- File Processing
- Exception handling
- Scope rules
- Containers
- File I/O
- Additional File Processing



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
```

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")
myFunction(34)
```

← This was imported by the command.

← This one wasn't!

Commonly Used Modules

- **Some useful modules to import, included with Python:**

- **Module: sys** **- Lots of handy stuff.**
  - sys.maxint
  - sys.argv
- **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?

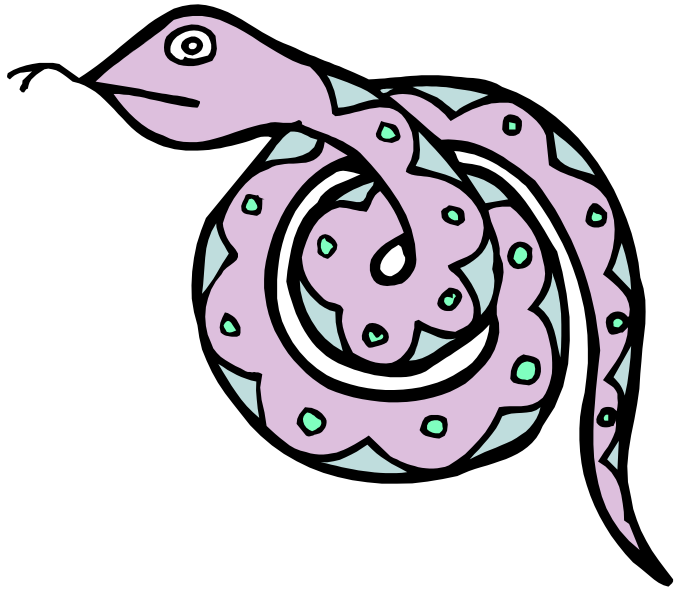
- **The list of directories in which Python will look for the files to be imported: sys.path**

  (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')
```

File Processing



File Processing with Python

This is a good way to play with the error handing capabilities of Python. Try accessing files without permissions or with non-existent names, etc.

You'll get plenty of errors to look at and play with!

```
import sys
fileptr = open('filename', 'r')
if fileptr == None:
     print("Something bad happened")
     sys.exit(1)
somestring = fileptr.read() # read one line
for line in fileptr:        # continue reading
   print (line)
fileptr.close()
```

Working with files

- **When opening a file, we can specify whether:**
  - We want to read or write content
  - We want to treat content as strings or as byte arrays

```
file = open("somefile", "r")    # read strings
file = open("somefile", "rb")   # read binary arrays

file = open("somefile", "w")    # write strings
file = open("somefile", "wb")   # write binary arrays
```

- **The choice has important consequences**
  - Regardless of how we process data in our Python code, we must be aware of how the file expects to read and write our data

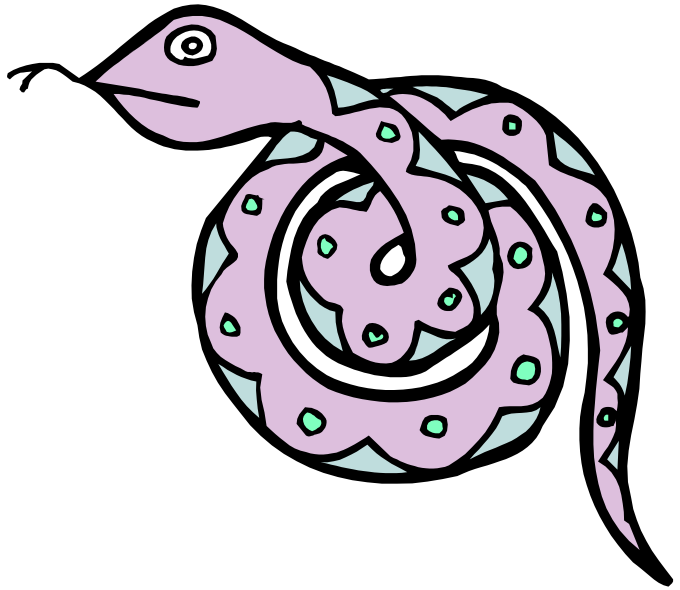
Working with files

```
>>> file = open("fubar", "w")
>>> file.write("hamberders\n")
11
>>> file.close()

>>> file = open("fubar.bin", "wb")
>>> file.write("hamberders")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' does not support the buffer interface
```

```
$ hexdump -C fubar
00000000  68 61 6d 62 65 72 64 65  72 73 0a                 |hamberders.|
0000000b
```

To work with binary files, use a specific library (e.g., pickle)

Exception Handling



Exception Handling

- **Errors are a kind of object in Python.**
  - More specific kinds of errors are subclasses of the general Error class.

- **You use the following commands to interact with them:**

  try

  except

  finally

Exceptions and handlers

```
while True:
   try:
        x = int(input("Number, please! "))
        print ("The number was: ", x)
        break
   except ValueError:
        print ("Oops! That was not a valid number.")
        print ("Try again.")
        print ()
```

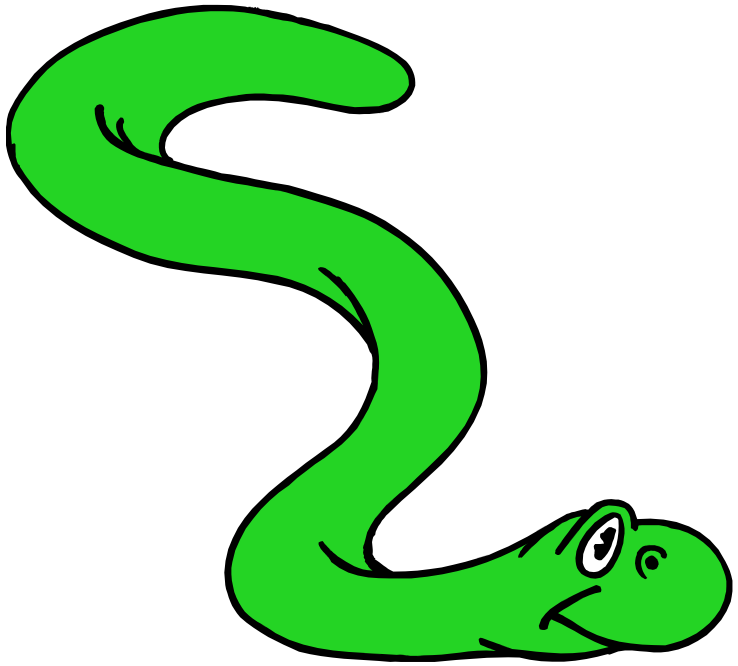
Exceptions and handlers

```
try:
   f = open("gift.txt", "r")
   # do file-open things here
   f.close()
except FileNotFoundError:
   print("Cannot open gift.txt")
   # do file-doesn't-exist stuff here
except PermissionError:
   print("Whatcha trying to do here with gift.txt, eh?")
   # do failure-due-to-permissions stuff here
```

Exceptions and handlers

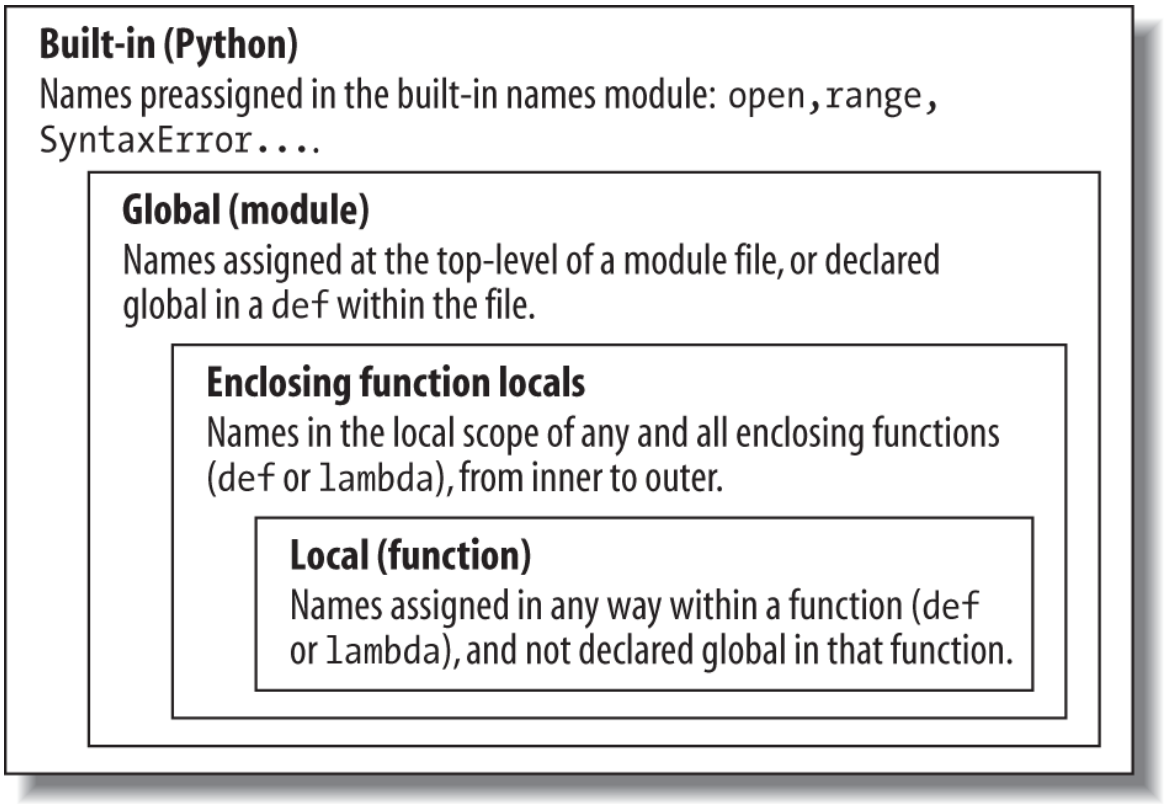
```
def loud_kaboom():
   x = 1/0;

def fireworks_factory():
   raise ZeroDivisionError("Gasoline near bone-dry Christmas trees!")

def playing_with_fire():
   try:
        loud_kaboom()
   except ZeroDivisionError as exc:
        print ("Handling run-time error: ", exc)

   try:
        fireworks_factory()
   except ZeroDivisionError:
        print ("Gotta stop this from happening...")
```

Scope Rules



Scope rules

- **In Java or C: scope of variable depends upon (a) location of declaration and possibly (b) extra modifiers**
- **In Python: scope depends upon (a) location of variable definition and possible (b) the "global" modifier**
- **LEGB rule used to resolve variable name:**
  - First search for Local definition...
  - ... and if not found there, go to Enclosing definition...
  - ... and if not found there go to "Global" definition...
  - ... and if not found there go to Built-in definition...
  - ... and if not found there, give up in despair.



from Figure 17-1, "Learning Python, Fifth edition", O'Reilly (2013)

Scope examples

```
X = 22

def func():
    X = 33

func()
print(X)  # Prints 22: unchanged
```

```
X = 99

def func(Y):
    Z = X + Y
    return Z

func(1)     # result is 100
```

Scope examples

```
X = 88

def func():
    global X
    X = 99

func()
print(X)   # Prints 99
```

```
y, z = 1, 2

def all_global():
        global x
        x = y + z    # LEGB indicate show to interpret y and z
```

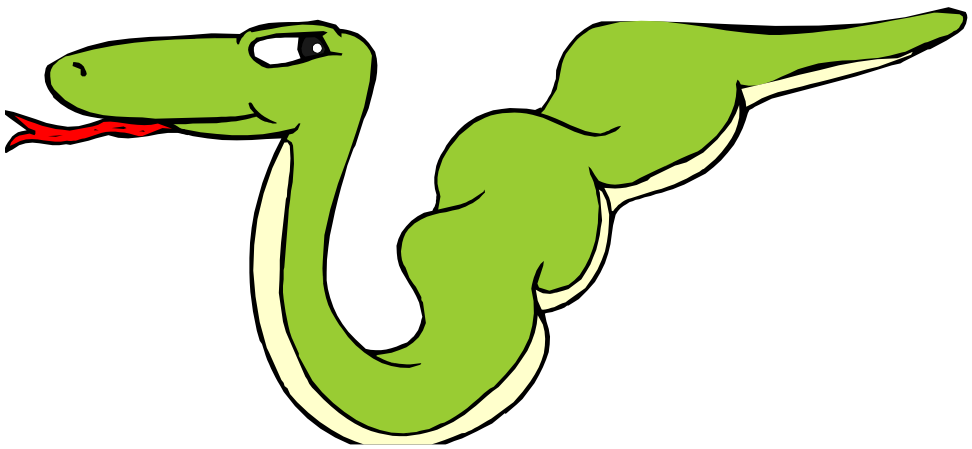
Nested function example (Python 3)

```
X = 99

def f1():
    X = 88
    def f2():
        print(X)   # LEGB rule indicates via "E" the meaning of X
    f2()

f1()   # Value output: 88
```

Scope rules

- **"Global" scope is actually "Module" scope**
- **Usually a good idea to minimize use of global variables**
  - Although sometimes it does make sense for a module to have its own global variables
  - Need to practice some judgement here
- **Not covered here (more complex):**
  - class scopes
  - comprehension scope
  - closures

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.)

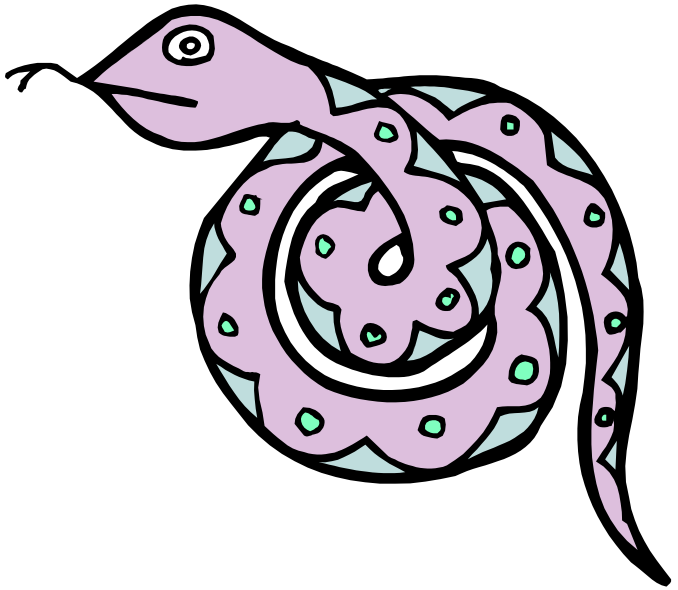
- **These three are empty, but of different *types***

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]
```

Additional File Processing



Writing specific ASCII values

- **This can be a bit tricky**
  - python supports both old-school ASCII (i.e., "latin-1" encodings) and more international UTF-8
  - Sometimes need to specify the encoding we want
  - Do not depend upon your platform defaults!

```
>>> file = open("oops.txt", "w")
>>> file.write("\xab\xba\xfe\xed")
4
>>> file.close()
```

```
$ hexdump -C oops.txt
00000000  c2 ab c2 ba c3 be c3 ed                           |........|
00000008
```

Writing specific ASCII values

```
>>> file = open("ver1.txt", encoding="latin-1", mode="w")
>>> file.write("\xab\xba\xfe\xed Gebürtstag")
15
>>> file.close()
```

```
$ hexdump -C ver1.txt
00000000  ab ba fe ed 20 47 65 62  fc 72 74 73 74 61 67     |???? Geb?rtstag|
0000000f
```

```
>>> file = open("ver2.txt", encoding="utf-8", mode="w")
>>> file.write("\xab\xba\xfe\xed Gebürtstag")
15
>>> file.close()
```

```
$ hexdump -C ver2.txt
00000000  c2 ab c2 ba c3 be c3 ad  20 47 65 62 c3 bc 72 74  |«°þí Gebürt|
00000010  73 74 61 67                                       |stag|
00000014
```

Working with binary

- **Sometimes we need the "binary" representation of some integer value**
  - We can do the conversion...
  - ... and use the result
- **It looks a little tricky**
  - There will be other ways of doing this, but here we want to convert the binary representation into something writable to a file

```
>>> import struct
>>> bytes = struct.pack("I", 121)    # 121 as a four-byte int
>>> chars = [chr(c) for c in bytes]  # each byte as a ASCII
>>> s = "".join(chars)
>>> s
'y\x00\x00\x00'
>>> somefile.write(s)
```

Working with binary

- And given some "binary" representation, we may want to get the original int

```
>>> import struct
...
>>> s = somefile.read(4)
>>> chars = list(s)
>>> ints = [ord(c) for c in chars]
>>> bytes = bytearray(ints)
>>> result = struct.unpack("I", bytes)
>>> result
(121,)
>>> result[0]
121
```

Options with file input

Sometimes we want to simply treat files as a sequence of bytes rather than as line of characters, but not used the packed form.

```
f = open("somefile.bin", encoding="latin-1", \
    mode ="r")


# Assuming open was successful
while True
     a_byte = f.read(1)    # read one char/byte
     if not a_byte:          # if byte == None
          break
     val = ord(a_byte)       # ASCII value of byte
     print(val)
f.close()
```