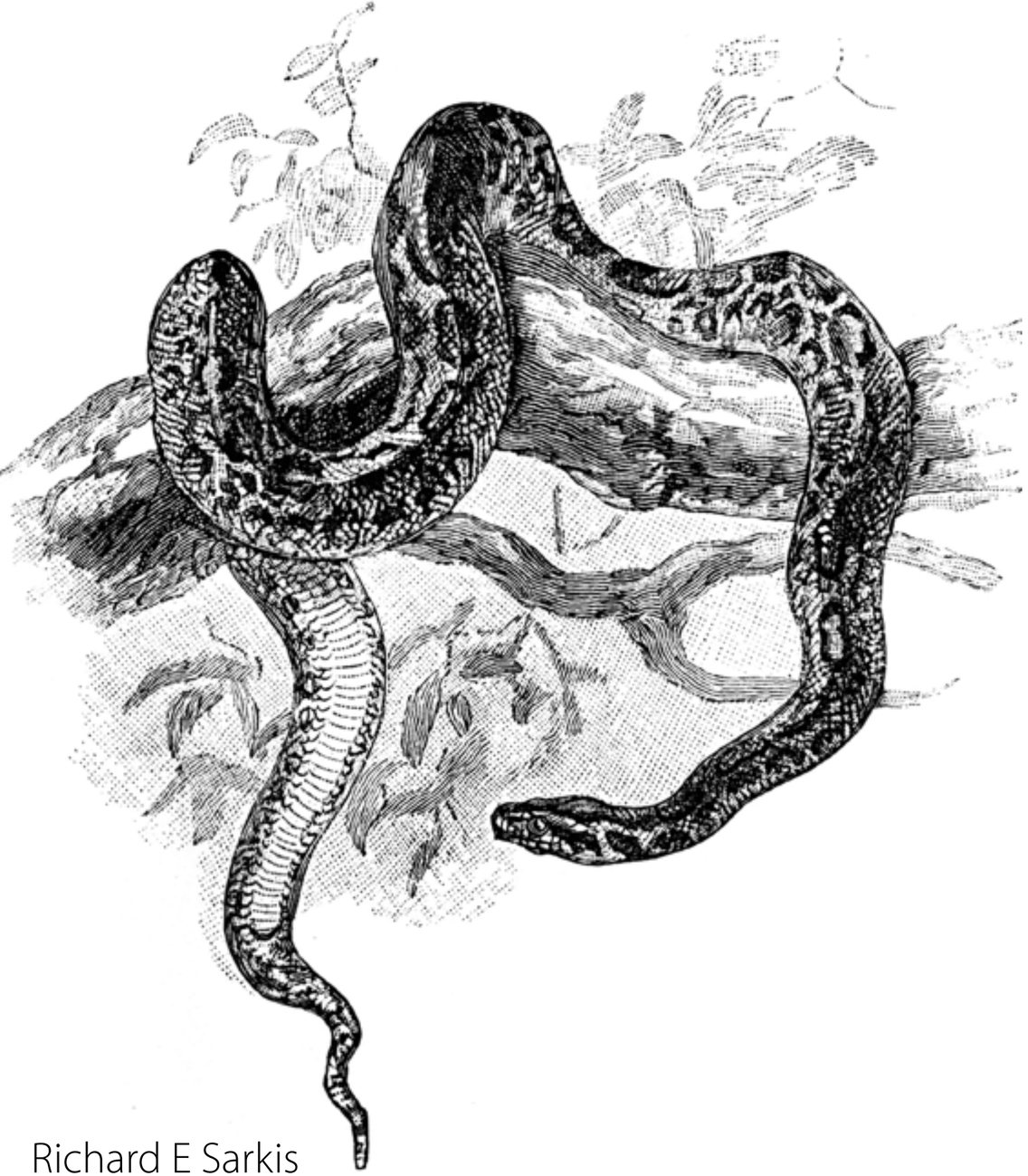

Wat's up, __doc__!?

Python "Gotchas"

Richard E Sarkis
Rochester's Python User Group
October 20th, 2015 Meeting

Pragmatic Mistakes

- Typing shell commands at the Interactive Prompt
- Print Statements are Required in Files (Only)
- Beware of Automatic Extensions on Windows
- Program-File Icon Click Pitfalls on Windows
- Imports Only Work the First Time
- Blank Lines Matter at the Interactive Prompt (Only)

Coding Mistakes

- Don't Forget the Colons
- Initialize Your Variables
- Start in Column 1
- Indent Consistently
- Always Use Parentheses to Call a Function
- Don't Use Extensions or Paths in Imports
- Don't Code C in Python

Common Programming Mistakes

Common Programming Mistakes

- **File-Open Calls Do Not Use the Module Search Path**
  - When you use the open() call in Python to access an external file, Python does not use the module search path to locate the target file. It uses an absolute path you give, or assumes the filename is relative to the current working directory. The module search path is consulted only for module imports.

Common Programming Mistakes

- **Methods Are Specific to Types**
  - You can't use list methods on strings, and vice versa. In general, methods calls are type-specific, but built-in functions may work on many types. For instance, the list reverse method only works on lists, but the len function works on any object with a length.

Common Programming Mistakes

- **Immutable Types Can't Be Changed in Place**
  - Remember that you can't change an immutable object (e.g., tuple, string) in place:

```
T = (1, 2, 3)
T[2] = 4  # Error
```

Common Programming Mistakes

- **Immutable Types Can't Be Changed in Place**
  - Construct a new object with slicing, concatenation, and so on, and assign it back to the original variable if needed. Because Python automatically reclaims unused memory, this is not as wasteful as it may seem:

```
T = T[:2] + (4,)
# Okay: T becomes (1, 2, 4)
```

Common Programming Mistakes

- **Use Simple for Loops Instead of while or range**

```
S = "lumberjack"

for c in S: print c                          # simplest

for i in range(len(S)): print S[i]    # too much

i = 0                                                 # too much
while i < len(S): print S[i]; i += 1
```

Common Programming Mistakes

- **Don't Expect Results From Functions That Change Objects**
  - In-place change operations such as the list.append( ) and list.sort( ) methods modify an object, but do not return the object that was modified (they return None); call them without assigning the result. It's not uncommon for beginners to say something like:

```
mylist = mylist.append(X)
```

Common Programming Mistakes

- **Don't Expect Results From Functions That Change Objects**
  - A more devious example of this pops up when trying to step through dictionary items in sorted-key fashion:

```
D = {...}
for k in D.keys().sort(): print D[k]
```

Common Programming Mistakes

- **Don't Expect Results From Functions That Change Objects**
  - This almost works -- the keys method builds a keys list, and the sort method orders it -- but since the sort method returns None, the loop fails because it is ultimately a loop over None (a nonsequence).

```
Ks = D.keys()
Ks.sort()
for k in Ks: print D[k]
```

Common Programming Mistakes

- **Conversions Only Happen Among Number Types**
  - In Python, an expression like 123 + 3.145 works -- it automatically converts the integer to a floating point, and uses floating point math. On the other hand, the following fails:

```
S = "42"
I = 1
X = S + I          # A type error
```

Common Programming Mistakes

- **Conversions Only Happen Among Number Types**
  - This is also on purpose, because it is ambiguous: should the string be converted to a number (for addition), or the number to a string (for concatenation)? In Python, we say that explicit is better than implicit (that is, EIBTI), so you must convert manually:

```
X = int(S) + I   # Do addition: 43
X = S + str(I)   # Do concatenation: "421"
```

Common Programming Mistakes

- **Cyclic Data structures Can Cause Loops**
  - Although fairly rare in practice, if a collection object contains a reference to itself, it's called a cyclic object. Python prints a [...] whenever it detects a cycle in the object, rather than getting stuck in an infinite loop:

```
>>> L = ['grail']  # Append reference back to L
>>> L.append(L)    # Generates cycle in object
>>> L
['grail', [...]]
```

Common Programming Mistakes

- **Assignment Creates References, Not Copies**

```
>>> L = [1, 2, 3]          # A shared list object
>>> M = ['X', L, 'Y']      # Embed a reference to
L
>>> M
['X', [1, 2, 3], 'Y']

>>> L[1] = 0               # Changes M too
>>> M
['X', [1, 0, 3], 'Y']
```

Common Programming Mistakes

- **Assignment Creates References, Not Copies**

```
>>> L = [1, 2, 3]
>>> M = ['X', L[:], 'Y']  # Embed a copy of L

>>> L[1] = 0              # Change only L, not M
>>> L
[1, 0, 3]
>>> M
['X', [1, 2, 3], 'Y']
```

Common Programming Mistakes

- **Local Names Are Detected Statically**
  - Python classifies names assigned in a function as locals by default; they live in the function's scope and exist only while the function is running. Technically, Python detects locals statically, when it compiles the defs code, rather than by noticing assignments as they happen at runtime. This can also lead to confusion if it's not understood.

Common Programming Mistakes

- **Local Names Are Detected Statically**

```
>>> X = 99
>>> def func():
...     print X       # Does not yet exist
...     X = 88        # Makes X local in entire
def
...
>>> func( )          # Error!
```

Common Programming Mistakes

- **Defaults and Mutable Objects**
  - Default argument values are evaluated and saved once, when the def statement is run, not each time the function is called. That's usually what you want, but since defaults retain the same object between calls, you have to be mindful about changing mutable defaults.

Common Programming Mistakes

- **Defaults and Mutable Objects**

```
>>> def saver(x=[]):    # Saves away a list object
...     x.append(1)     # and changes it each time
...     print x
...
>>> saver([2])          # Default not used
[2, 1]
>>> saver()             # Default used
[1]
>>> saver()             # Grows on each call!
[1, 1]
>>> saver()
[1, 1, 1]
```

Common Programming Mistakes

- **Defaults and Mutable Objects**
  - Some see this behavior as a feature -- because mutable default arguments retain their state between function calls, they can serve some of the same roles as static local function variables in the C language. However, this can seem odd the first time you run into it, and there are simpler ways to retain state between calls in Python (e.g., classes)

Common Programming Mistakes

- **Defaults and Mutable Objects**

```
>>> def saver(x=None):
...     if x is None: x = []   # No arg passed?
...     x.append(1)            # Changes new list
...     print x
...
>>> saver([2])                 # Default not used
[2, 1]
>>> saver()                    # Doesn't grow now
[1]
>>> saver()
[1]
```

Common Programming Mistakes

- **Using class variables incorrectly**

```
>>> class A(object):
...     x = 1
...
>>> class B(A):
...     pass
...
>>> class C(A):
...     pass
...
>>> print A.x, B.x, C.x
1 1 1
```

Common Programming Mistakes

- **Misunderstanding Python scope rules**

```
>>> x = 10
>>> def foo():
...     x += 1
...     print x
...
>>> foo()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in foo
UnboundLocalError: local variable 'x' referenced
before assignment
```

Common Programming Mistakes

- **Modifying a list while iterating over it**

```
>>> odd = lambda x : bool(x % 2)
>>> numbers = [n for n in range(10)]
>>> for i in range(len(numbers)):
...     if odd(numbers[i]):
...         del numbers[i]  # BAD: Deleting item
from a list while iterating over it
...
Traceback (most recent call last):
      File "<stdin>", line 2, in <module>
IndexError: list index out of range
```

Wat

Converting to a string and back

```
>>> bool(str(False))
???
```

Mixing integers with strings

```
>>> int(2 * 3)
???
>>> int(2 * '3')
???
>>> int('2' * 3)
???
```

The undocumented converse implication operator

```
>>> False ** False == True
???
>>> False ** True == False
???
>>> True ** False == True
???
>>> True ** True == True
???
```

Mixing numerical types

```
>>> x = (1 << 53) + 1
>>> x + 1.0 < x
???
```

Operator precedence?

```
>>> False == False in [False]
???
```

Iterable types in comparisons

```
>>> a = [0, 0]
>>> (x, y) = a
>>> (x, y) == a
???

>>> [1,2,3] == sorted([1,2,3])
???
>>> (1,2,3) == sorted((1,2,3))
???
```

Types of arithmetic operations

```
>>> type(1) == type(-1)
???
>>> 1 ** 1 == 1 ** -1
???
>>> type(1 ** 1) == type(1 ** -1)
???
```

Fun with iterators

```
>>> a = 2, 1, 3
>>> sorted(a) == sorted(a)
???
>>> reversed(a) == reversed(a)
???

>>> b = reversed(a)
>>> sorted(b) == sorted(b)
???
```

Circular types

```
>>> isinstance(object, type)
???
>>> isinstance(type, object)
???
```

extend vs +=

```
>>> a = ([],)
>>> a[0].extend([1])
>>> a[0]
???
>>> a[0] += [2]
???
>>> a[0]
???
```

Indexing with floats

```
>>> [4][0]
???
>>> [4][0.0]
???
>>> {0:4}[0]
???
>>> {0:4}[0.0]
???
```

all and emptiness

```
>>> all([])
???
>>> all([[]])
???
>>> all([[[]]])
???
```

sum and strings

```
>>> sum("")
???
>>> sum("", ())
???
>>> sum("", [])
???
>>> sum("", {})
???
>>> sum("", "")
???
```

Comparing NaNs

```
>>> x = 0*1e400  # nan
>>> len({x, x, float(x), float(x), 0*1e400, 0*1e400})
???
>>> len({x, float(x), 0*1e400})
???
```

References

- When Pythons Attack: Common Mistakes of Python Programmers.
- Python wats.
- Become More Advanced: Master the 10 Most Common Python Programming Mistakes.
- Wat's up, doc?