
# ECE 364
# Software Engineering Tools Lab

Lecture 8

Python: Advanced I

# Lecture Summary

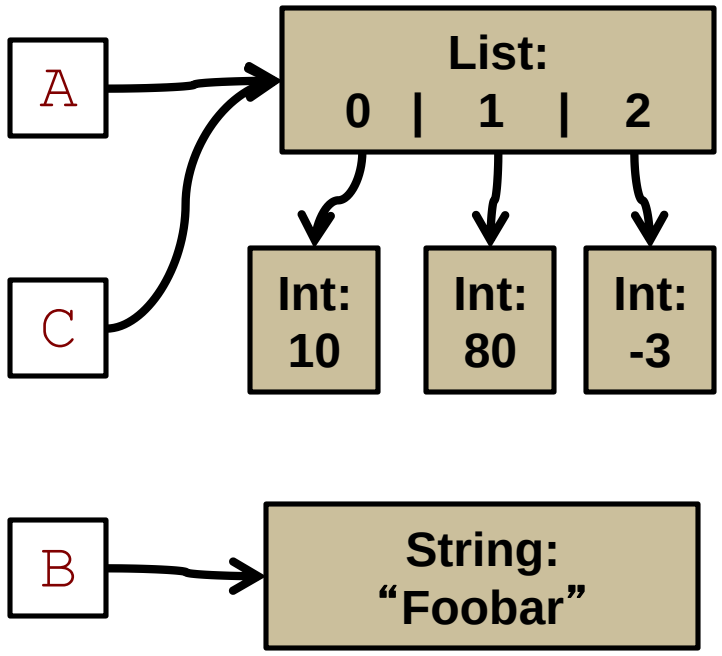
- Python Variables
- Namespaces and Scope
- Modules
- Exceptions

# More on Python Variables

- All variables in Python are actually pointers

```
>>> A = [10, 80, -3]
>>> B = "Foobar"
>>> C = A
```

- This is represented in memory like this:




# More on Python Variables

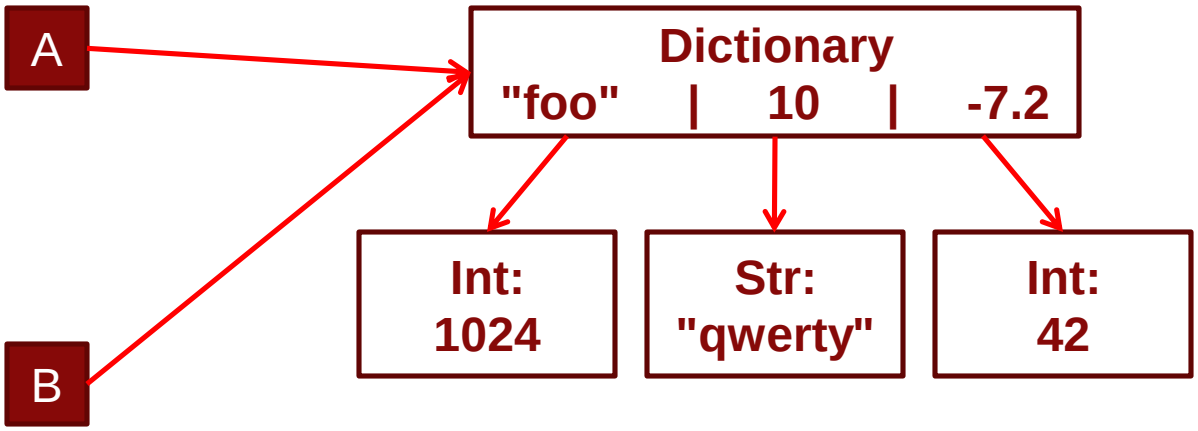
- Whenever you use **=**, you are reassigning a pointer, not making a copy or altering a value.
- This can lead to side effects
- Example:

```
>>> A = [10, 80, -3]
>>> B = A
>>> B.append('Z')
>>> print(A)
[10, 80, -3, 'Z']
```

# Making Copies

- The assignment `B = A` only points `B` to `A`

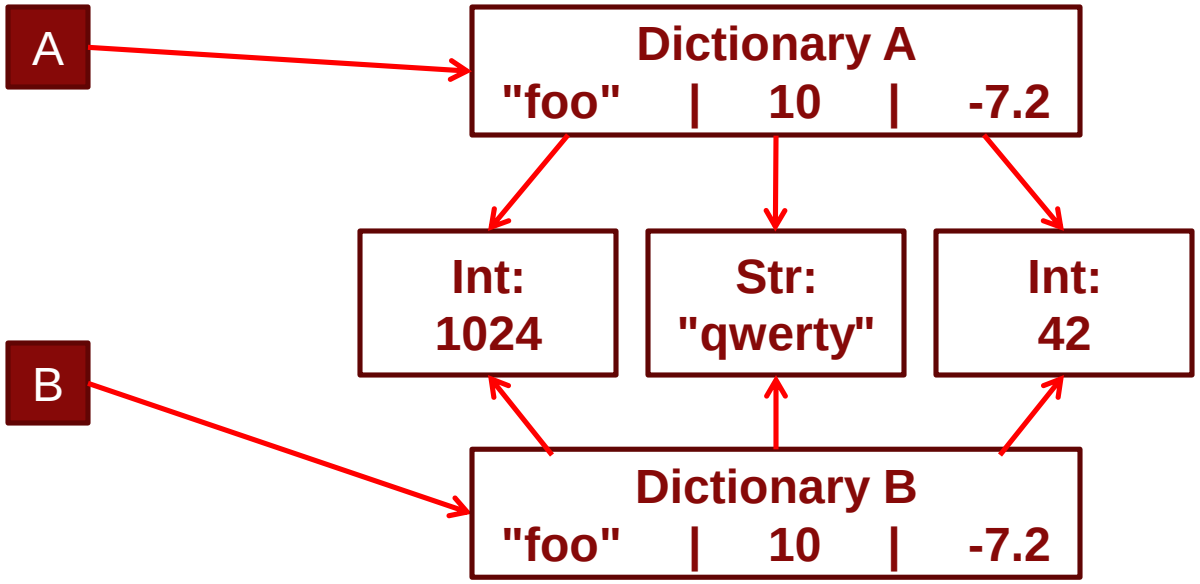
```
A={"foo":1024, 10:"qwerty", -7.2:42}
```



# Making Copies (Shallow)

- Use the `copy()` function
- A shallow copy only duplicates the keys!

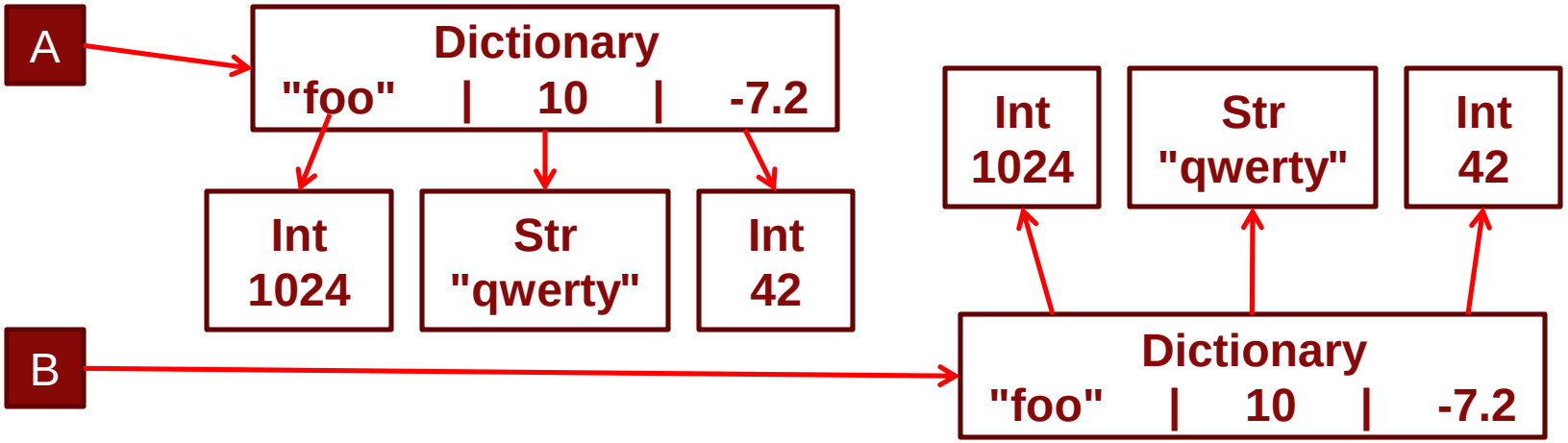
```
B = A.copy()
```




# Making Copies (Deep)

- The copy module provides `deepcopy()`
- Copies keys and values

```
import copy
B = copy.deepcopy(A)
```



# Namespaces

- A namespace is "a mapping from names to objects"
  - e.g. "`x`" maps to the value 5 or "`A`" maps to [1, 2, 3]

- Python creates namespaces at different times
  - The built-in namespace is created when Python starts and is never deleted
  - A namespace is created for a function when the function is called and deleted when the function returns

# Scope

- A scope is "a textual region of a Python program where a namespace is directly accessible"

- A new local namespace is created whenever a function is called

- Python resolves identifiers by searching their current scope in the following order
  1. Local namespace
  2. Global namespace within the module
  3. “Built-in" namespace
  4. If the name can not be resolved a NameError exception is raised

# Scope (2)

```
Y=0                         # X and Y are part of the __global__ namespace
X=0

def foo():
    X = 10                          # X and Y are part of a new namespace
    Y = 40                          # __global__.foo, and are distinct from
    print("{}, {}".format(X, Y))# X and Y in the global namespace

def bar():
    X = 30                          # Python will search for names within
    print("{}, {}".format(X, Y))# the local namespace upwards to the
                                    # __global__ namespace. Y is found _
                                    # _global__

print("{}, {}".format(X, Y))        # prints 0 0
foo()                               # prints 10 40
print("{}, {}".format(X, Y))        # prints 0 0
bar()                               # prints 30 0
print("{}, {}".format(X, Y))        # prints 0 0
```

# Comparison: Scope in C

- Scope in C is defined by blocks
  - Created using curly braces

```
void foo(void *bar) {
    int i = 0;
    while (bar[i] != '\0') {
        i++;
    }
    return i;
}
```

# Global Variables

- To write to a variable that is in any namespace other than the current, local namespace you must declare it as global in the current namespace

```
def ChangeA():
  global A
  A = 25

A = 404
ChangeA ()
print(A)

>>> 25
```

```
def ChangeA():
  A = 25


A = 404
ChangeA ()
print(A)

>>> 404
```

# Modules

- A python module is a file containing function definitions and statements that enables code reuse and provides modular structure to your programs
  - See `Pro_Set.py` in the example scripts

- An `import` statement loads the module and makes it's functions and variables visible to the current namespace
  - `sys` and `os` are commonly used modules

- A module's name is specified by it's file name
  - e.g. The module `Pro_Set` exists in the file `Pro_Set.py`

# Modules (2)

- A module must be imported into the current namespace using an `import` statement
  - `import mod_name`

- When accessing a function or variable from the module you must prepend the module name
  - e.g. **`os.`**`access(…)` or **`sys.`**`argv`

# Modules (3)

- You can omit the module name when referring to functions or variables defined in the module when using an alternative import style
  - `from mod_name import func_name`

- The `from` style import makes the function or variable name directly visible to the current name space

# Modules (4)

```
from sys import argv, stderr
if len(argv) != 2:
  stderr.write("usage: script.py <arg1>\n")
```

- `argv` and `stderr` do not have a `sys.` prefix because they are directly imported from the sys module into the current namespace

# Modules (5)

- An asterisk in the `from` style import allows you to import all functions and variables in the module into the current namespace

```
from sys import *
if len(argv) != 2:
  stderr.write("usage: script.py <arg1>\n")
```

- You should avoid using this for more than one module
  - It clutters the namespace and may cause name conflicts

# Modules (6)

- Modules can be imported almost anywhere within a python program
  - Typically done at the very beginning of a file
  - But can be done elsewhere…

```
def print_usage():
   import sys # sys only visible in print_usage()
   sys.stdout.write("usage: %s\n" % (sys.argv[0],))
```

- When importing into a lower namespace the module is only accessible from within that namespace

# What is an Exception?

- An exception is a program branch executed when an invalid operation or state is detected
- Each statement in a python program my cause an exception to occur, immediately changing the execution path
- Exceptions are used in many other high level languages
  - C++, Java, C# etc.
- In C there are no exceptions
  - Programmer must resort to using error codes and other mechanisms

# What is an Exception?

- An exception is the error
  - Python is literally throwing the error at you
  - You can either catch it or let it knock you out

- Exceptions contain helpful details about what went wrong in the program
  - Error messages, function name, argument values etc.
  - Much more useful than error code

- Not as simple as an error code but the number 10 does not tell much about what went wrong

# Built-In Exceptions

- `ValueError` – indicates a value is not properly formed
- `IOError` – indicates a problem when preforming I/O
- `OSError` – indicates an error raised by the operating system
- `IndexError` – indicates an index lookup failed or was out of bounds
- `KeyError` – indicates an lookup by key failed or the key was not found
- `TypeError`– indicates an function or operation is applied to a type that does not support it

# Handling Exceptions

- There are two things a program can do with exceptions
  - Ignore them – will result in early termination
  - Catch them – examine the exception and perform some action to correct the problem

- It is not always possible to correct errors at runtime
  - But you can display a user friendly error message and possibly log the error

```
try:
   … statements …
except:
   … exception handling statements …
```

# Handling Exceptions (2)

- There are many different types of exceptions.
- Many times you will want to handle them in a different way depending on the type.

```
try:
   … statements that may raise an exception …
except (IOError, ValueError):
   … handle IO errors or value errors …
except <ExType> as e:
   … handle <ExType> error referenced as the variable e…
except:
   … handle all other errors …
else:
   … statements to run if no exception was raised …
finally:
   … Always run after handling error …
```

# Handling Exceptions (4)

- `except IOError:`
  - No reference to the `IOError` exception is available.
- `except IOError as e:`
  - The `IOError` exception and its details are available in the local variable `e`.
  - Newer syntax that is preferred for new programs.
- `except IOError, e:` (In Python 2)
  - Equivalent to the above statement.

# Handling Exceptions (5)

```
import sys

file_name = raw_input('Enter a file name: ')

fp = open(File_name, "r")  # possible exception

sum = 0.0
for line in fp: # possible exception
    values = line.split(',')
    for v in values:
        fv = float(v) # possible exception
        sum += fv

fp.close()

print "Sum = %f" % (sum,)
sys.exit(0)
```

Take branch if exception raised

Take branch if exception raised

Take branch if exception raised

Built-in Exception Handler:

```
import sys, traceback

t, v, tb = sys.exc_info()
traceback.print_exception(t, v, tb, limit=2, file=sys.stderr)
sys.exit(-1)
```

# Handling Exceptions (6)

```
file_name = raw_input('Enter a file name: ')

try:
    fp = open(file_name, "r") # possible exception
except:
    print "Could not open %s" (file_name,)
    sys.exit(1)

sum = 0.0
for line in fp: # possible exception
    values = line.split(',')
    for v in values:
        try:
            fv = float(v) # possible exception
            sum += fv
        except ValueError as e:
            print "Unknown value %s: %s" % (v,e)

fp.close()

print "Sum = %f" % (sum,)
sys.exit(0)
```

Take branch if exception raised

Take branch if exception raised

Take branch if exception raised

Built-in Exception Handler:

```
t, v, tb = sys.exc_info()
traceback.print_exception(t, v, tb, limit=2, file=sys.stderr)
sys.exit(-1)
```

# Handling Exceptions (7)

```
value = raw_input('Prompt: ')

try:
     # int() Raises a ValueError exception
     result = proc_integer(int(val))
except ValueError:
     result = proc_other(val)

print(value)
```

# Handling Exceptions (8)

- To ignore an exception the `pass` statement can be used to indicate not action is taken

```
value = raw_input('Prompt: ')

try:
  result = proc_integer(int(val))
except ValueError:
  pass # do nothing…

print(value)
```

# Handling Exceptions (9)

```
import sys

try:
     fp = open(sys.argv[1], "r")
except IOError as e:
     sys.stderr.write("Reason: %s\n" % (e,))
     sys.exit(1)

for line in fp:
     print(line)

fp.close()
sys.exit(0)
```

# Raising Exceptions

- There are often many times when you want to signal an error to your program
- Typically you will raise exceptions from your own modules and functions to indicate errors or malformed arguments
- Exceptions are “raised” by the program using the `raise <Exception>` statement

# Raising Exceptions (2)

```
def dotProduct(row, column):
    if type(row) is not list or type(column) is not list:
        raise TypeError("Row & Column must be lists.")

    if ((len(row) == len(column)) and (len(row) > 0)):
        raise ValueError("Row & Column must be of the same size.")

    # Continue execution.
    …

row2 = [3, 7, 1, 8]; column2 = [2, 5, 6]

# These will raise an exception.
dotProduct(row2, column2)
dotProduct(9, column2)
```