Introduction to Python

Introduction to Software-defined Networking
Block Course – 16-20 March 2015

David Koll

Based on slides of Matt Huenerfauth from the University of Pennsylvania

Today's Afternoon Schedule

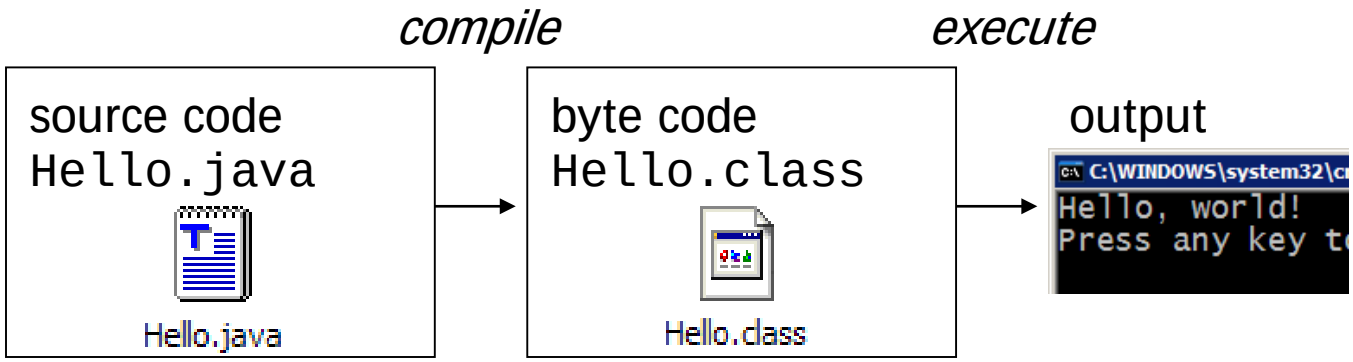
- Roughly 90 minutes:
  - Introduction to Python
    - Basic syntax, concepts, …
- Later: Installing Python (if required)
- Even later: First exercises in Python (Homework 2)

What is Python?

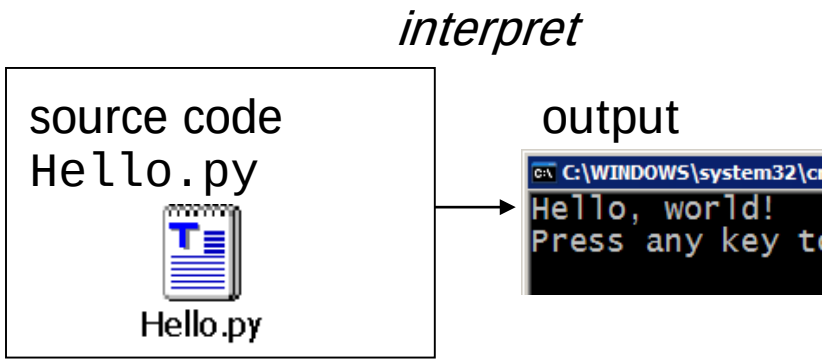
- An *interpreted* programming language
  - …with strong similarities to PERL
  - …with powerful typing and object oriented features.
  - …with useful built-in types (lists, dictionaries).
  - …with clean syntax, powerful extensions.

Compiling and Interpreting

- Many languages require compiling



- Python is directly *interpreted* into machine instructions.



Compiling and Interpreting

- Compiled languages are usually faster at runtime
  - Code is already translated to machine code
- Interpreted languages are usually faster in development
  - No need to compile before testing
- We will see the implications on SDN later in this course

Why Python in this course?

- Reason: Mininet
- Mininet is a emulation/virtualization tool to develop, test and deploy (software-defined) networks
- Written in…Python
  - Has an API that you have to understand for this course!

How will we teach Python?

- We expect that you are familiar with object oriented programming
  - We expect that you know at least one of C++/JAVA…
  - …and how to use an IDE (Eclipse, Emacs, NetBeans …)
- We expect that you know the concepts of:
  - Data types
  - Variable declaration and assignment
  - Control sequences (if/else, while, for, …)
  - Functions
  - Classes
- We will focus on the syntax and special issues of Python here! It is sort of a crash course…

A word on the Exercises

- You will be programming in Python in several exercises
- Some are easy, some a bit more difficult
- Rule of thumb: Easier exercises are due earlier, you have more time for the more difficult ones
- Exercise 2: 10/04/2015
- Exercise 6: 17/04/2015
- Other programming exercises: 30/04/2015
- However: try to keep up!

A word on the Exercises

- If you have questions:
  - Try Google
  - Ask your fellow students personally
  - During the course, use the exercise slots to ask me
  - Use the SDN course mailing list
  - Send me an e-mail

Technical Issues

Installing & Running Python

Some words ahead

- We will be using Python 2.7
  - For differences to Python 3.4 (newer version)
  - In short:
    - 2.7 is default version on most UNIX OS distributions (e.g. Ubuntu 14)
    - 2.7 has better library support
  - Note that Python 3 will be the future standard (but differences are not that big)

Installing Python

- Python for Windows from www.python.org.
- For UNIX: No installation should be required
- GUI development environments:
  - Eclipse (PyDev)
  - Emacs
  - IDLE.
  - Your favorite IDE or text editor!

Running Interactively on UNIX

On Unix…

```
% python

>>> 3+3
6
```

The '>>>' is the Python prompt.

Running Programs on UNIX

```
% python filename.py
```

You can create python files using emacs.
(There's a special Python editing mode.)
Want to make *.py file executable? Add on top:

```
#!/pkg/bin/python
```

UNDERSTANDING THE BASICS

Look at a sample of code…

```
x = 34 - 23                 # A comment.
y = “Hello”                 # Another one.
z = 3.45
if z == 3.45 or y == “Hello”:
    x = x + 1
    y = y + “ World”   # String concat.
print x
print y
```

Look at a sample of code…

```
x = 34 - 23                    # A comment.
y = “Hello”                    # Another one.
z = 3.45
if z == 3.45 or y == “Hello”:
    x = x + 1
    y = y + “ World”   # String concat.
print x
print y
```

Enough to Understand the Code

- Assignment uses = and comparison uses ==.
- First assignment to a variable will create it.
  - Variable types don't need to be declared.
    - Python figures out the variable types on its own.
- For numbers +-*/% are as expected.
  - Special use of + for string concatenation.
  - Special use of % for string formatting.
- Logical operators are words (**`and`**, **`or`**, **`not`**) *not symbols (&&, ||, !).*

Whitespace

- Whitespace is meaningful in Python: ***especially indentation and placement of newlines.***
  - Use a newline to end a line of code. *(Not a semicolon like in C++ or Java.)*
    - (Use \ when must go to next line prematurely.)
  - No braces { } to mark blocks of code in Python…
    - Use consistent *indentation* instead.
    - The first line with a new indentation is considered outside of the block.
  - Often a colon appears at the start of a new block.

Basic Datatypes

- Integers (default for numbers)

  z = 5 / 2 # Answer is 2, integer division.

- Floats

  x = 3.456

- Strings
  - Can use “” or ‘’ to specify. “abc” ‘abc’ (Same thing.)
  - Unmatched ones can occur within the string. “matt’s”
  - Use triple double-quotes as kind of escape-character: “““a‘b“c”””

Look at a sample of code…

```
x = 34 - 23                    # A comment.
y = “Hello”                    # Another one.
z = 3.45
# colon indicates new block
if z == 3.45 or y == “Hello”:
   #indentation signals new block
   x = x + 1
   y = y + “ World”   # String concat.
#indentation signals block end
print x
print y
```

Comments

- Start comments with # – the rest of line is ignored.
- Convention: “documentation string”
  - in the first line of any new function or class that you define.

```
def my_function(x, y):
    """This is the docstring. This
    function does blah blah blah."""
    # The code would go here...
```

Look at a sample of code…

```
x = 34 - 23                    # A comment.
y = “Hello”                    # Another one.
z = 3.45
if z == 3.45 or y == “Hello”:
    x = x + 1
    y = y + “ World”           # String concat.
print x
print y
```

Understanding Assignment

Names and References 1

- Python has no pointers like C or C++.
  - Instead, it has “names” and “references”. (Works a lot like Java.)

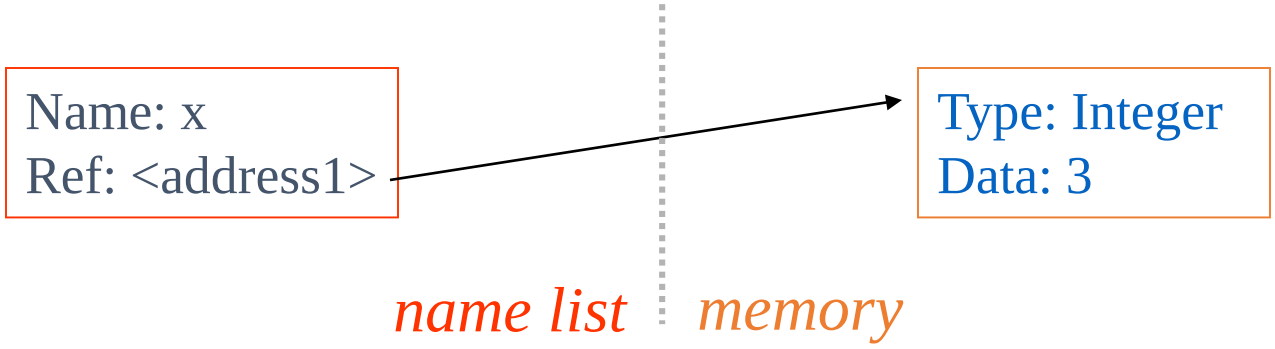
- You create a name the first time it appears on the left side of an assignment expression:

```
x = 3
```

- Names store references
  - Python determines the type of the reference automatically based on what data is assigned to it.
  - It also decides when to delete it via garbage collection after any names for the reference have passed out of scope.

Names and References 2

- **x = 3**
  - an integer 3 is created and stored in memory.
  - Then, a name x is created.
  - Reference to the memory location storing the 3 is then assigned to the name x.



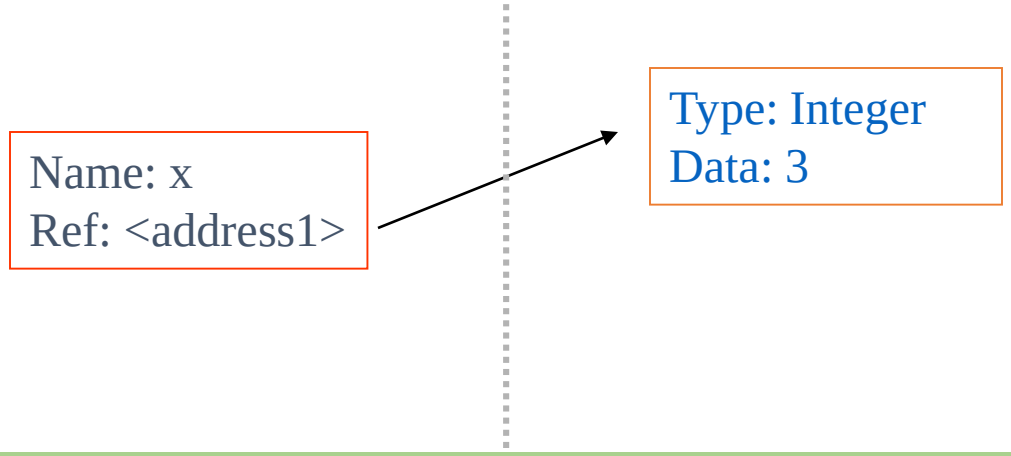
Names and References 3

- The data 3 we created is of type integer. In Python, the basic datatypes integer, float, and string are "immutable."
- This doesn't mean we can't change the value of x… For example, we could increment x.

```
>>> x = 3
>>> x = x + 1
>>> print x
4
```

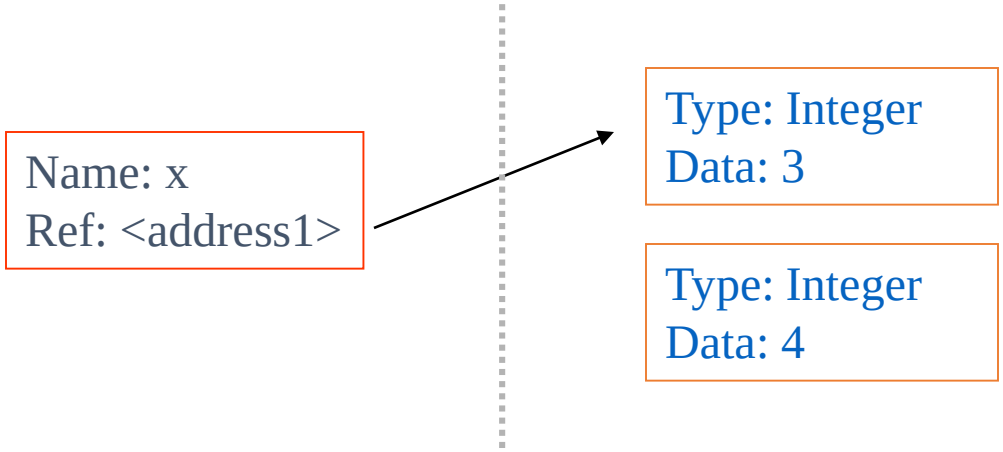
Names and References 4

- If we increment x, then what's really happening is:
  - The reference of name x is looked up.
  - The value at that reference is retrieved.
  - The 3+1 calculation occurs, producing a new data element 4 which is assigned to a fresh memory location with a new reference.
  - The name x is changed to point to this new reference.
  - The old data 3 is garbage collected if no name still refers to it.



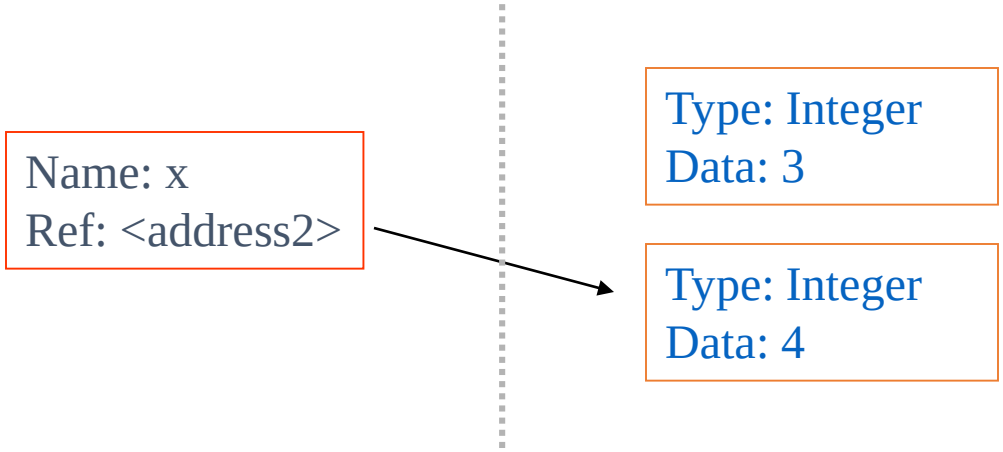
Names and References 4

- If we increment x, then what's really happening is:
  - The reference of name x is looked up.
  - The value at that reference is retrieved.
  - The 3+1 calculation occurs, producing a new data element 4 which is assigned to a fresh memory location with a new reference.
  - The name x is changed to point to this new reference.
  - The old data 3 is garbage collected if no name still refers to it.



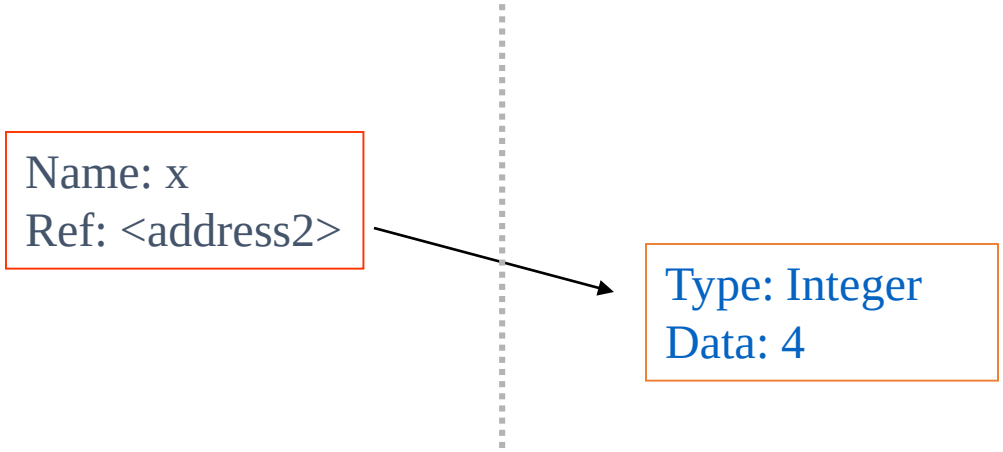
Names and References 4

- If we increment x, then what's really happening is:
  - The reference of name x is looked up.
  - The value at that reference is retrieved.
  - The 3+1 calculation occurs, producing a new data element 4 which is assigned to a fresh memory location with a new reference.
  - The name x is changed to point to this new reference.
  - The old data 3 is garbage collected if no name still refers to it.



Names and References 4

- If we increment x, then what's really happening is:
  - The reference of name x is looked up.
  - The value at that reference is retrieved.
  - The 3+1 calculation occurs, producing a new data element 4 which is assigned to a fresh memory location with a new reference.
  - The name x is changed to point to this new reference.
  - The old data 3 is garbage collected if no name still refers to it.



Assignment 2

- For other data types (lists, dictionaries, user-defined types), assignment works differently.
  - These datatypes are **"mutable."**
  - When we change these data, we do it *in place.*
  - We don't copy them into a new memory address each time.
  - If we type y=x and then modify y, both x and y are changed!
  - We'll talk more about "mutability" later.

immutable

```
>>> x = 3
>>> y = x
>>> y = 4
>>> print x
3
```

mutable

x = some mutable object

y = x

make a change to y

look at x

x will be changed as well

Multiple Assignment

- You can also assign to multiple names at the same time.

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

Naming Rules

- Names are case sensitive and cannot start with a number. They can contain letters, numbers, and underscores.

  bob Bob _bob _2_bob_ bob_2 BoB

- There are some reserved words:

```
and, assert, break, class, continue,
def, del, elif, else, except, exec,
finally, for, from, global, if, import,
in, is, lambda, not, or, pass, print,
raise, return, try, while
```

Container Types in Python

Container Types

- Containers are built-in data types in Python.
  - Can hold objects of any type (including their own type).
  - There are three kinds of containers:

Tuples

- A simple immutable ordered sequence of items.

Lists

- Sequence with more powerful manipulations possible.

Dictionaries

- A look-up table of key-value pairs.

Contents can be of varying type!

Tuples, Lists, and Strings 1

- Tuples are defined using parentheses (and commas).

```
>>> tu = (23, 'abc', 4.56, (2,3), 'def')
```

- Lists are defined using square brackets (and commas).

```
>>> li = ["abc", 34, 4.34, 23]
```

- Strings are defined using quotes (", ', or """).

```
>>> st = "Hello World"
>>> st = 'Hello World'
>>> st = """This is a multi-line
string that uses triple quotes."""
```

Tuples, Lists, and Strings 2

- We can access individual members of a tuple, list, or string using square bracket “array” notation.

```
>>> tu[1]      # Second item in the tuple.
 'abc'
>>> li[1]       # Second item in the list.
 34
>>> st[1]   # Second character in string.
 'e'
```

Similar Syntax

- Tuples and lists are sequential
  - Share a lot of syntax
    - Most operations shown in this section can be applied to both
- Strings are essentially lists of characters
  - Operations you see in this section apply to them as well.

Looking up an Item

```
>>> t = (23, 'abc', 4.56, (2,3), 'def')
```

Positive index: count from the left, starting with 0.

```
>>> t[1]
'abc'
```

Negative lookup: count from right, starting with –1.

```
>>> t[-3]
4.56
```

Slicing: Return Copy of a Subset 1

```
>>> t = (23, 'abc', 4.56, (2,3), 'def')
```

Return a copy of the container with a subset of the original members. Start copying at the first index, and stop copying before the second index.

```
>>> t[1:4]
('abc', 4.56, (2,3))
```

You can also use negative indices when slicing.

```
>>> t[1:-1]
('abc', 4.56, (2,3))
```

Slicing: Return Copy of a Subset 2

```
>>> t = (23, 'abc', 4.56, (2,3), 'def')
```

Omit the first index to make a copy starting from the beginning of the container.

```
>>> t[:2]
(23, 'abc')
```

Omit the second index to make a copy starting at the first index and going to the end of the container.

```
>>> t[2:]
(4.56, (2,3), 'def')
```

Copying the Whole Container

You can make a copy of the whole tuple using **[:]**.

```
>>> t[:]
(23, 'abc', 4.56, (2,3), 'def')
```

So, there's a difference between these two lines:

```
>>> list2 = list1   # 2 names refer to 1 ref
                    # Changing one affects both

>>> list2 = list1[:]   # Two copies, two refs
                       # They're independent
```

The ‘in’ Operator

- Boolean test whether a value is inside a container:

```
>>> t = [1, 2, 4, 5]
>>> 3 in t
 False
>>> 4 in t
 True
>>> 4 not in t
 False
```

- Be careful: the ‘in’ keyword is also used in the syntax of other unrelated Python constructions: *for loops* and *list comprehensions*.

The + Operator

- The + operator produces a new tuple, list, or string whose value is the concatenation of its arguments.

```
>>> (1, 2, 3) + (4, 5, 6)
 (1, 2, 3, 4, 5, 6)

>>> [1, 2, 3] + [4, 5, 6]
 [1, 2, 3, 4, 5, 6]

>>> “Hello” + “ ” + “World”
 ‘Hello World’
```

The * Operator

- The * operator produces a new tuple, list, or string that “repeats” the original content.

```
>>> (1, 2, 3) * 3
(1, 2, 3, 1, 2, 3, 1, 2, 3)

>>> [1, 2, 3] * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]

>>> “Hello” * 3
‘HelloHelloHello’
```

Mutability: Tuples vs. Lists

Tuples: Immutable

```
>>> t = (23, 'abc', 4.56, (2,3), 'def')
>>> t[2] = 3.14

Traceback (most recent call last):
  File "<pyshell#75>", line 1, in -toplevel-
    tu[2] = 3.14
TypeError: object doesn't support item assignment
```

Not allowed to change a tuple *in place* in memory

OK to make a fresh tuple and assign its reference to a previously used name.

```
>>> t = (1, 2, 3, 4, 5)
```

Lists: Mutable

```
>>> li = ['abc', 23, 4.34, 23]
>>> li[1] = 45
>>> li
 ['abc', 45, 4.34, 23]
```

We can change lists *in place.*

`li` still points to the same memory reference when we're done.

Operations on Lists Only 1

- Many more operations can be performed on (mutable) lists than on (immutable) tuples.

- But: lists not as fast as tuples.
  - Trade-off.

Operations on Lists Only 2

```
>>> li = [1, 2, 3, 4, 5]

>>> li.append('a')
>>> li
[1, 2, 3, 4, 5, 'a']

>>> li.insert(2, 'i')
>>>li
[1, 2, 'i', 3, 4, 5, 'a']
```

Operations on Lists Only 3

```
>>> li.extend([9, 8, 7])
>>>li
[1, 2, 'i', 3, 4, 5, 'a', 9, 8, 7]
```

- '+' vs 'extend'?
  - + creates a fresh list (with a new memory reference)
  - extend operates on list **`li`** in place.
- Extend takes a list as an argument. Append takes a singleton.

```
>>> li.append([9, 8, 7])
>>> li
[1, 2, 'i', 3, 4, 5, 'a', 9, 8, 7, [9, 8, 7]]
```

Operations on Lists Only 4

```
>>> li = ['a', 'b', 'c', 'b']

>>> li.index('b')      # index of first occurrence
1

>>> li.count('b')      # number of occurrences
2

>>> li.remove('b')     # remove first occurrence
>>> li
  ['a', 'c', 'b']
```

Operations on Lists Only 5

```
>>> li = [5, 2, 6, 8]

>>> li.reverse()    # reverse the list *in place*
>>> li
  [8, 6, 2, 5]

>>> li.sort()       # sort the list *in place*
>>> li
  [2, 5, 6, 8]
```

Tuples vs. Lists

- Lists slower but more powerful than tuples.
  - Lists can be modified, and they have lots of handy operations we can perform on them.
  - Tuples are immutable and have fewer features.

- We can always convert between tuples and lists using the list() and tuple() functions.

```
li = list(tu)
tu = tuple(li)
```

GENERATING LISTS USING "LIST COMPREHENSIONS"

List Comprehensions

- A powerful feature of the Python language.
  - Generate a new list by applying a function to every member of an original list.

- The syntax of a “list comprehension” is different from what you may have seen so far

List Comprehensions Syntax 1

```
>>> li = [3, 6, 2, 7]
>>> [elem*2 for elem in li]
[6, 12, 4, 14]
```

[expression **for** name **in** list **]**

- Where expression is some calculation or operation acting upon the variable name.

- Results are stored in a new list!

List Comprehension Syntax 2

- Also possible on containers:

```
>>> li = [('a', 1), ('b', 2), ('c', 7)]
>>> [ n * 3 for (x, n) in li]
[3, 6, 21]
```

List Comprehension Syntax 3

- The expression of a list comprehension could also contain user-defined functions.

```
>>> def subtract(a, b):
        return a - b

>>> oplist = [(6, 3), (1, 7), (5, 5)]

>>> [subtract(y, x) for (x, y) in oplist]
[-3, 6, 0]
```

Filtered List Comprehension 1

[expression **for** name **in** list **if** filter **]**

- Exclude some list members from applying comprehension

- First check each member of the list to see if it satisfies a filter condition.

Filtered List Comprehension 2

```
[ expression for name in list if filter ]
```

```
>>> li = [3, 6, 2, 7, 1, 9]
>>> [elem * 2 for elem in li if elem > 4]
[12, 14, 18]
```

- Only 6, 7, and 9 satisfy the filter condition.

Nested List Comprehensions

- Nested comprehensions possible.

```
>>> li = [3, 2, 4, 1]
>>> [elem*2 for elem in
        [item+1 for item in li] ]
[8, 6, 10, 4]
```

- The inner comprehension produces: [4, 3, 5, 2].
- The outer comprehension produces: [8, 6, 10, 4].

DICTIONARIES

Basic Syntax for Dictionaries 1

- Dictionaries store a mapping between a set of keys and a set of values.
  - Keys can be any immutable type.
  - Values can be any type, ***and you can have different types of values in the same dictionary.***

- You can define, modify, view, lookup, and delete the key-value pairs in the dictionary.

Basic Syntax for Dictionaries 2

```
>>> d = {'user':'bozo', 'pswd':1234}

>>> d['user']
'bozo'

>>> d['pswd']
1234

>>> d['bozo']

Traceback (innermost last):
  File '<interactive input>' line 1, in ?
KeyError: bozo
```

Basic Syntax for Dictionaries 3

```
>>> d = {'user':'bozo', 'pswd':1234}

>>> d['user'] = 'clown'
>>> d
{'user':'clown', 'pswd':1234}
```

Note: Keys are unique.
Assigning to an existing key just replaces its value.

```
>>> d['id'] = 45
>>> d
{'user':'clown', 'id':45, 'pswd':1234}
```

Note: Dictionaries are unordered.
New entry might appear anywhere in the output.

Basic Syntax for Dictionaries 4

```
>>> d = {'user':'bozo', 'p':1234, 'i':34}

>>> del d['user']                  # Remove one.
>>> d
{'p':1234, 'i':34}

>>> d.clear()                      # Remove all.
>>> d
{}
```

Basic Syntax for Dictionaries 5

```
>>> d = {'user':'bozo', 'p':1234, 'i':34}

>>> d.keys()                  # List of keys.
['user', 'p', 'i']

>>> d.values()                # List of values.
['bozo', 1234, 34]

>>> d.items()       # List of item tuples.
[('user','bozo'), ('p',1234), ('i',34)]
```

Assignment and Containers

Assignment & Mutability 1

- Remember that assignment works differently for mutable vs. immutable datatypes.
  - If you type y=x, then changing y:
    …will change x if they are mutable.
    …won't change x if they are immutable.

immutable

```
>>> x = 3
>>> y = x
>>> y = y + 1
>>> print x
3
```

mutable

```
>>> x = [ 1, 2, 3]
>>> y = x
>>> y.reverse()
>>> print x
[ 3, 2, 1]
```

Multiple Assignment with Container Classes

- We've seen multiple assignment before:

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

- But you can also do it with containers.
  - The type and "shape" just has to match.

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

Empty Containers 1

- We know that assignment is how to create a name.

  `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.)

Control of Flow

If Statements

```
if x == 3:
    print “X equals 3.”
elif x == 2:
    print “X equals 2.”
else:
    print “X equals something else.”
print “This is outside the ‘if’.”
```

While Loops

```
x = 3
while x < 10:
    x = x + 1
    print “Still in the loop.”
print “Outside of the loop.”
```

`break` and `continue` just like in other languages

Assert

- An assert statement will check to make sure that something is true during the course of a program.
  - program stops otherwise.

```
assert(number_of_players < 5)
```

The Loop Else Clause

- The optional **else** clause runs only if the loop exits normally (not by break)

```
x = 1

while x < 3 :
    print x
    x = x + 1
else:
    print 'hello'
```

In whileelse.py

```
~: python whileelse.py
1
2
hello
```

Run from the command line

The Loop Else Clause

```
x = 1
while x < 5 :
    print x
    x = x + 1
    break
else :
    print 'i got here'
```

whileelse2.py

```
~: python whileelse2.py
1
```

For Loops / List Comprehensions

- List comprehensions usually require a for-loop in other programming languages.
  - For loops are rarer in Python!

.

For Loops 1

- On any “iteratable” object:

```
for <item> in <collection>:
    <statements>
```

- When <collection> is a list or a tuple, then the loop steps through each element of the container.
- When <collection> is a string, then the loop steps through each character of the string.

```
for someChar in “Hello World”:
    print someChar
```

For Loops 2

- <item> can also be more complex
  - elements of a container <collection> are also containers,
    - <item> part of the for loop can match the structure of the elements.
  - This multiple assignment can make it easier to access the individual parts of each element.

```
for (x, y) in [(a,1), (b,2), (c,3), (d,4)]:
  print x
```

range() function

- range() returns a list of numbers from 0 up to the number we pass to it (non-inclusive).
- range(5) returns [0,1,2,3,4]

```
for x in range(5):
    print x
```

- range([start], stop[, step])

`range(4:10:2)` returns 4,6,8

- Can also go backwards (**`range(4:-4:-2)`**)

LOGICAL EXPRESSIONS

True and False

- True and False are constants in Python.
  - Generally, True equals 1 and False equals 0.
- Other values equivalent to True and False:
  - False: zero, None, empty container or object
  - True: non-zero numbers, non-empty objects.

Boolean Logic Expressions

- You can also combine Boolean expressions.
  - True if a is true and b is true: `a and b`
  - True if a is true or b is true: `a or b`
  - True if a is false: `not a`

- Use parentheses as needed to disambiguate complex Boolean expressions.

String Operations

String Operations

- We can use some methods built-in to the string data type to perform some formatting operations on strings:

```
>>> “hello”.upper()
‘HELLO’
```

- There are many other handy string operations available. Check the Python documentation for more.

String Formatting Operator: %

- The operator % allows building a string out of multiple data items in a “fill in the blanks” fashion.
  - similar to the printf command of C.

Formatting Strings with %

```
>>> x = "abc"
>>> y = 34
>>> "%s xyz %d" % (x, y)
'abc xyz 34'
```

- The tuple following the % operator is used to fill in the blanks in the original string marked with %s or %d.
  - Check Python documentation for whether to use %s, %d, or some other formatting code inside the string.

Printing with Python

- "print"
- Example with % string operator:

```
>>> print  "%s xyz %d"  %  ("abc", 34)
abc xyz 34
```

- "Print" automatically adds a newline at end of the string.
- Concatenation of strings with a space between them.

```
>>> print "abc" >>> print "abc", "def"
abc                abc def
```

Type Conversions

String to List to String

- Works like in other languages

```
>>> x = 4          # python will create int
>>> y = float(4)   # 4.00
>>> z = str(4)     # “4”
```

- See documentation for details

String to List to String

- Join turns a list of strings into one string.

  <separator_string>.join(<some_list>)

```
>>> ";".join( ["abc", "def", "ghi"] )
 "abc;def;ghi"
```

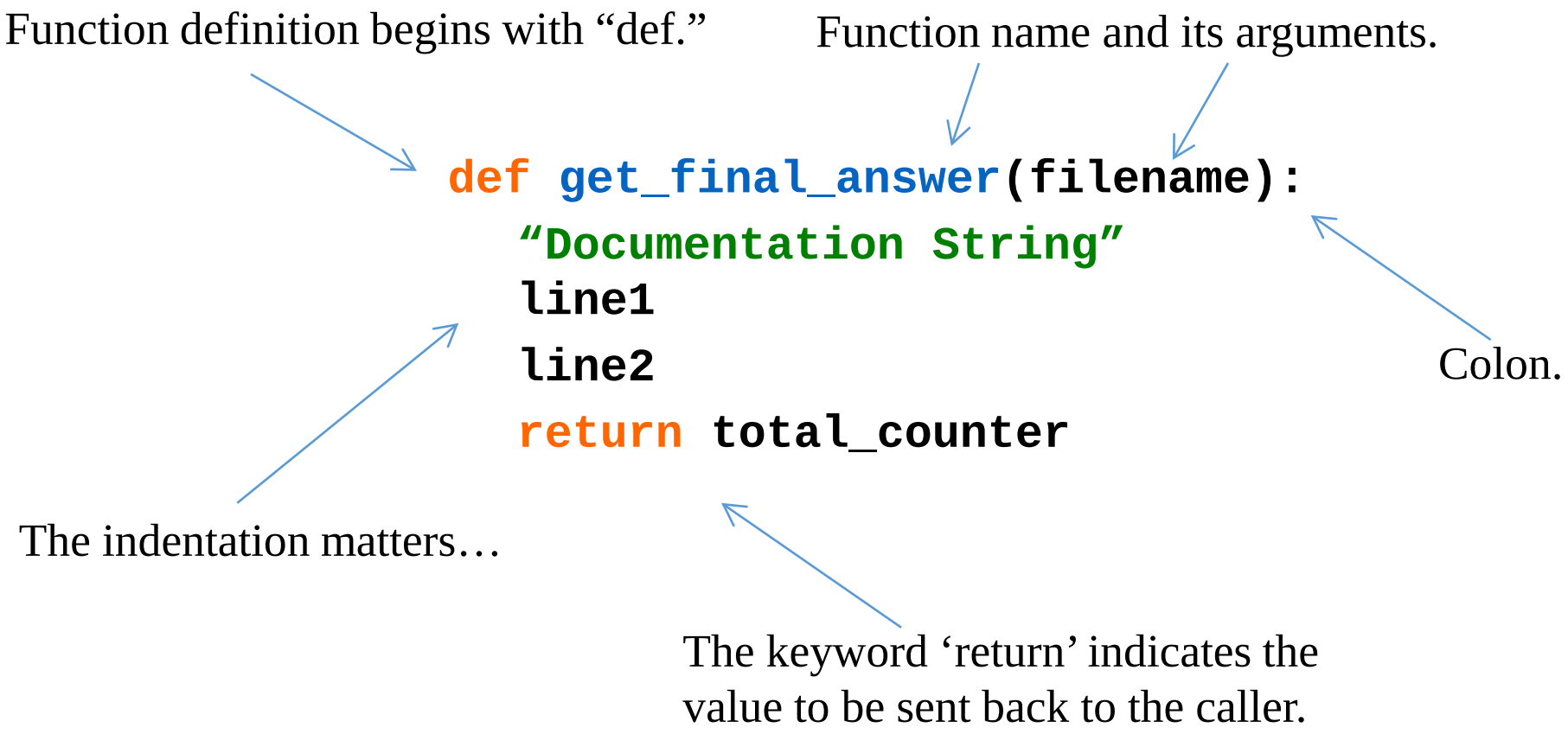
- Split turns one string into a list of strings.

  <some_string>.split(<separator_string>)

```
>>> "abc;def;ghi".split( ";" )
 ["abc", "def", "ghi"]
```

FUNCTIONS IN PYTHON

Defining Functions



No header file or declaration of <u>types</u> of function or arguments.

Calling a Function

- The syntax for a function call is:

```
>>> def myfun(x, y):
        return x * y
>>> myfun(3, 4)
12
```

- Parameters in Python are “Call by Assignment.”
  - Sometimes acts like “call by reference” and sometimes like “call by value” in C++.
    - Mutable datatypes: Call by reference.
    - Immutable datatypes: Call by value.

Functions without returns

- ***All*** functions in Python have a return value
- Functions without a “return” will give the special value **None** as their return value.
  - None is used like NULL, void, or nil in other languages.
  - Also logically equivalent to False.

Function overloading? No.

- There is no function overloading in Python.
  - Unlike C++, a Python function is specified by its name alone
  - the number, order, names, or types of its arguments cannot be used to distinguish between two functions with the same name.

You can't have two functions with the same name, even if they have different arguments.

Treating Functions Like Data

- Functions are treated like first-class objects in the language… They can be passed around like other data and be arguments or return values of other functions.

```
>>> def myfun(x):
        return x*3

>>> def applier(q, x):
        return q(x)

>>> applier(myfun, 7)
21
```

Assignment & Mutability

- When passing parameters to functions:
  - Immutable data types are **"call by value."**
  - Mutable data types are **"call by reference."**

If you pass mutable data to a function, and you change it inside that function, the changes will persist after the function returns.

Immutable data appear unchanged inside of functions to which they are passed.

Some Fancy Function Syntax

Lambda Notation – Anonymous Functions

- Sometimes it is useful to define short functions without having to give them a name: especially when passed as an argument to another function.

```
>>> applier(lambda z: z * 4, 7)
28
```

  - First argument to applier() is an unnamed function that takes one input and returns the input multiplied by four.
  - Note: only single-expression functions can be defined using this lambda notation.

Default Values for Arguments

- Give default values for a function's arguments when defining it
  - these arguments are optional when function is called.

```
>>> def myfun(b, c=3, d="hello"):
        return b + c
>>> myfun(5,3,"hello")
>>> myfun(5,3)
>>> myfun(5)
```

All of the above function calls return 8.

The Order of Arguments

```
>>> def myfun(a, b, c):
        return a-b
>>> myfun(2, 1, 43)
  1
>>> myfun(c=43, b=1, a=2)
  1
>>> myfun(2, c=43, b=1)
  1
```

Object Oriented Programming in Python

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()
```

  - Looks like Java or C++ method calls.

Defining Classes

Defining a Class

- Python doesn't use separate class interface definitions as in some languages.
  - You just define the class and then use it.

- You can define a method in a class by including function definitions within the scope of the class block.
  - Take care of proper indentation!

Definition of student

```
#class definition
class student:
    """A class representing a
    student."""
    #init function?
    #self argument?
    def __init__(self,n,a):
        self.full_name = n
        self.age = a

    def get_age(self):
        return self.age
```

Constructor: __init__

- **`__init__`** acts like a constructor for a class.
  - Invoked upon instantiating a class
  - **`b = student("Bob", 21)`**

    __init__ is passed "Bob" and 21.

Constructor: __init__

- **`__init__`** can take any number of arguments.

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

Self

- The first argument of every class method is a reference to the current instance of the class.
  - By convention, this argument is named **`self`**.
- In **`__init__`**, self refers to the object currently being created
- 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)
```

Creating and Deleting Instances

Instantiating Objects

- There is no “new” keyword as in Java.
- You merely use the class name with () notation and assign the result to a variable.

```
b = student(“Bob Smith”, 21)
```

- The arguments you pass to the class name are actually given to its `.__init__()` method.

- User-defined classes are mutable!

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.
  - 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
```

ATTRIBUTES

Two Kinds of Attributes

- The non-method data stored by objects are called attributes. There's two kinds:
  - **Data attribute:**
    - Variable owned by a particular instance of a class.
  - **Class attributes:**
    - Owned by the class as a whole.
    - Called "static" variables in some languages.
    - Good for class-wide constants or for building counter of how many instances of the class have been made.

Data Attributes

- inside of the **`__init__()`** method.
  - Inside the class, refer to data attributes using **`self`** – for example, **`self.full_name`**

```
class teacher:
    "A class representing teachers."
    def __init__(self,n):
        self.full_name = n
    def print_name(self):
        print self.full_name
```

Class Attributes

- All instances of a class share one copy of a class attribute
  - if any of the instances changes it, value is changed for all instances.
- Define class attributes outside of any method.
- Access them using **`self.__class__.name`** notation.

```
class sample:
x = 23
def increment(self):
    self.__class__.x += 1
```

```
>>> a = sample()
>>> a.increment()
>>> a.__class__.x
24
```

Data vs. Class Attributes

```
class counter:
 overall_total = 0
       # class attribute
 def __init__(self):
      self.my_total = 0
        # data attribute
 def increment(self):
     counter.overall_total = \
     counter.overall_total + 1
     self.my_total = \
     self.my_total + 1
```

```
>>> a = counter()
>>> b = counter()
>>> a.increment()
>>> b.increment()
>>> b.increment()
>>> a.my_total
1
>>> a.__class__.overall_total
3
>>> b.my_total
2
>>> b.__class__.overall_total
3
```

INHERITANCE

Subclasses

- Inheritance works pretty much like in other languages
  - New class: “subclass.” Original: “parent” or “ancestor.”
- Syntax of defining a subclass:

```
class subclass(parent):
```

e.g.:

```
class student(person):
```

- Python has no ‘extends’ keyword like Java.
- Multiple inheritance is supported.

Definition of student

```
class person:
 "A class representing a person."

 def __init__(self,n,a):
     self.full_name = n
     self.age = a

 def get_age(self):
     return self.age
```

Definition of ai_student

```
class student (person):
 “A class extending person.”

 def __init__(self,n,a,number):
      student.__init__(self,n,a)
      self.mat_num = number

 def get_age():
      print “Age: ” + str(self.age)
```

Redefining Methods

- Overwrite parent class methods if desired.

- Can still explicitly call the parent's version of the method.

```
parentClass.methodName(self, a, b, c)
```

The only time you ever explicitly pass 'self' as an argument is when calling a method of an ancestor.

Redefining the __init__

- Same as for redefining any other method…
  - You'll often see something like this in the **`__init__`** method of subclasses:

```
parentClass.__init__(self, x, y)
```

Special Built-In Methods and Attributes

Built-In Members of Classes

- All built-in members have double underscores around their names: **`__init__`** **`__doc__`**

Special Methods

- For example, the method **`__repr__`** exists for all classes, and you can always redefine it.
- The definition of this method specifies how to turn an instance of the class into a string.
  - When you type **`print f`** you are really calling **`f.__repr__()`** to produce a string for object f.
  - If you type **`f`** at the prompt and hit ENTER, then you are also calling **`__repr__`** to determine what to display to the user as output.

Special Methods – Example

```
class person:
  ...
   def __repr__(self):
     return "I'm named " + self.full_name
  ...
```

```
>>> f = student("Bob Smith", 23)
>>> print f
I'm named Bob Smith
>>> f
"I'm named Bob Smith"
```

Special Methods

- Possible to redefine:
  - **`__init__`** : The constructor for the class.
  - **`__cmp__`** : Define how **`==`** works for class.
  - **`__len__`** : Define how **`len(`** obj **`)`** works.
  - **`__copy__`** : Define how to copy a class.
- Other built-in methods allow you to give a class the ability to use [ ] notation like an array or ( ) notation like a function call.

Special Data Items

- These attributes exist for all classes.

  `__doc__` : Variable storing the documentation string for that class.

  `__class__` : Variable which gives you a reference to the class from any instance of it.

  `__module__` : Variable which gives you a reference to the module in which the particular class is defined.

Special Data Items – Example

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

>>> print f.__doc__
A class representing a student.

>>> f.__class__
< class studentClass at 010B4C6 >
```

Private Data and Methods

- Any attribute or method with two **leading underscores** is private.
  - Note:
    Names with two underscores **at the beginning and the end** are for built-in methods or attributes for the class.
  - Note:
    There is no ‘protected’ status in Python; so, subclasses would be unable to access these private data either.

Importing and Modules

Importing and Modules

- Use classes & functions defined in another file.
- Like Java import, C++ include.
- Three formats of the command:

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

What's the difference?

What gets imported from the file and what name you use to refer to it after its been imported.

import …

```
import somefile
```

- Everything in somefile.py gets imported.
- Access:

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

from … import *

```
from somefile import *
```

- Everything in somefile.py gets imported.
- You can just use the name of the stuff in the file. It's all brought into the current namespace.
- Be careful when using this import command! You could overwrite the definition of something in your file!

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

from … import …

```
from somefile import className
```

- Only one item in somefile.py gets imported.
- You can just use the name of the item in the file. It's brought into the current namespace.
- Be careful when using this import command! You could overwrite the definition of this one name in your file!

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

← This got imported.

← This one didn't.

Common Modules

- Here are some modules you might like to import. They come included with Python…

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

Common Modules

- math - Math stuff
  - Exponents
  - sqrt
- Random - Randomization.
  - Randrange
  - Uniform
  - Choice
  - Shuffle

Defining your own modules

- You can save your own code files (modules) and import them into Python using the import commands we learned.

Where do you put them?

What directory do you save module files so Python can find them and import them?

- 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.)

- If you want to add a directory of your own to this list, you need to append it to this list.

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

I/O

Input

- The **raw_input**(string) method returns a line of user input as a string
- The parameter is used as a prompt
- The string can be converted by using the conversion methods **int**(string), **float**(string), etc.

Input: Example

```
print "What's your name?"
name = raw_input("> ")

print "What year were you born?"
birthyear = int(raw_input("> "))

print "Hi %s! You are %d years old!" % (name, 2011 - birthyear)
```

```
~: python input.py
What's your name?
> Michael
What year were you born?
>1980
Hi Michael! You are 31 years old!
```

File Handling

inflobj = open(‘data’, ‘r’)	Open the file ‘data’ for input
S = inflobj.read()	Read whole file into one String
S = inflobj.read(N)	Reads N bytes (N >= 1)
L = inflobj.readlines()	Returns a list of line strings

Files: Output

outflobj = open(‘data’, ‘w’)	Open the file ‘data’ for writing
outflobj.write(S)	Writes the string S to file
outflobj.writelines(L)	Writes each of the strings in list L to file
outflobj.close()	Closes the file

Example: Read a file and print line by line

```
fileptr = open('filename')
somestring = fileptr.read()
for line in fileptr:
    print line
fileptr.close()
```

Remember to close all opened files!

FINALLY: ERROR 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
  - Catch
- Works pretty much like in JAVA

Exception Handling

```
#!/usr/bin/python

try:
     fh = open("testfile", "w")
     try:
          fh.write("This is a test")
     finally:
          print "Going to close the file"
          fh.close()
except IOError:
     print "Error: can\'t find file or read data"
```