

CSCI 553: Networking III

Unix Network Programming

Spring 2007



Strings, Lists and Files



Today's Agenda

- UNP Task for Today
- Question about Lab 03
 - Repositories for source files only (no object or executable files)
- More Python Scripting
 - Strings
 - List
 - Functions
- Lab 04 (@ 10:15 pm)
 - Finish 4.2 today



Introduction

- To recap:
 - Python is a nimble language
 - Ideal for building tools and crunching data
 - Has the usual data types and control flow constructs
- This lecture describes:
 - Strings
 - Lists
 - File I/O
- After this lecture, you will be able to use Python to crunch simple data formats



Strings

- A string is an *immutable sequence of characters*
- *Sequence* means that it can be indexed
 - Indices start at 0 (as in C, Java, and C#)
 - So `text[0]` is the first character of `text`
- The built-in function `len` returns the length of a string
 - So the last character of `text` has index `len(text)-1`

```
element = "boron"
i = 0
while i < len(element):
    print element[i]
    i += 1
b o r o n
```
- **Note:** there is no separate data type for characters
 - A character is simply a string of length 1



Immutability

- ***Immutable*** means that it cannot be modified once it has been created
 - I.e., you cannot change individual characters in place

```
$ python
>>> element = 'gold'
>>> print 'element is', element
element is gold
>>> element[0] = 's'
TypeError: object does not support item assignment
```
- Why?
 - Safety (which we'll discuss in **Sets, Dictionaries, and Complexity**)
 - Doesn't actually cost that much, since most changes to strings change their length
- Of course, you *can* assign a new string value to a variable

```
>>> element = 'gold'
>>> print 'element is', element
element is gold
>>> element = 'lead'
>>> print 'element is now', element
element is now lead
```

Slicing

- `text[start:end]` takes a *slice* out of text
 - Creates a new string containing the characters of text from start up to (but not including) end
- ```
element = "helium"
print element[1:3], element[:2], element[4:]
el he um
```
- Sometimes helps to think of indices as being *between* elements

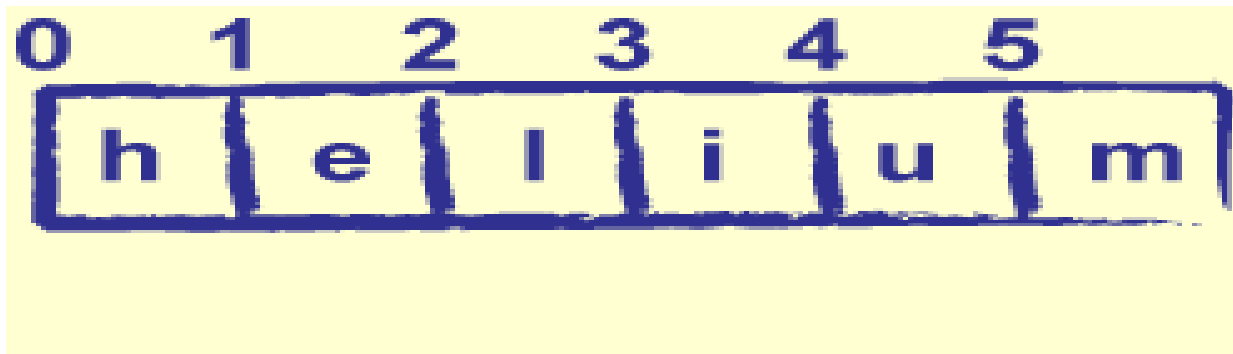


Figure 7.1: Visualizing Indices



# Bounds Checking

---

- Python always does an out-of-bounds check when you index a single item

- But it truncates out-of-range indices when you take a slice

```
$ python
```

```
>>> element = 'helium'
```

```
>>> print element[1:22]
```

```
elium
```

```
>>> x = element[22]
```

```
IndexError: string index out of range
```

- *"A foolish consistency is the hobgoblin of little minds."* (Ralph Waldo Emerson)

# Negative Indices

- Negative indices count backward from the end of the string
    - `x[-1]` is the last character
    - `x[-2]` is the second-to-last character
- ```
element = "carbon"  
print element[-2], element[-4], element[-6]  
o r c
```
- A lot easier to read than `x[len(x)-1]`
 - Again, it helps to visualize the indices as lying *between* the characters

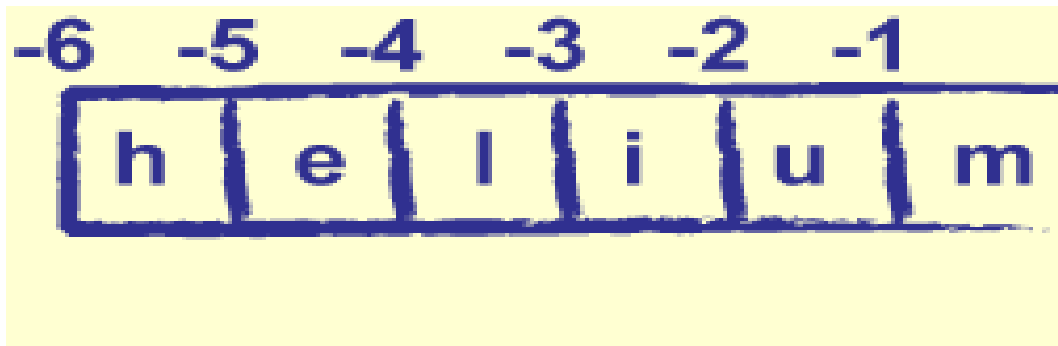


Figure 7.2: Visualizing Negative Indices



Consequences

- `text[1:2]` is either:
 - The second character in `text`...
 - ...or the empty string (if `text` doesn't have a second character)
- So `text[2:1]` is always the empty string
- So is `text[1:1]`
 - From index 1, up to but not including index 1
- `text[1:-1]` is everything except the first and last characters
 - Which may again be the empty string



Methods

- A *method* is a function that is tied to a particular *object*
 - Invented to help programmers organize their code
 - We'll see how to create objects and methods of our own *later*
- Almost everything in Python has methods
 - Numbers are the only important exception
- To call a method `meth` of object `obj`, type `obj.meth()`



String Methods

Method	Purpose	Example	Result
capitalize	Capitalize first letter of string	<code>"text".capitalize()</code>	<code>"Text"</code>
lower	Convert all letters to lowercase	<code>"aBcD".lower()</code>	<code>"abcd"</code>
upper	Convert all letters to uppercase	<code>"aBcD".upper()</code>	<code>"ABCD"</code>
strip	Remove leading and trailing whitespace (blanks, tabs, newlines, etc)	<code>" a b ".strip()</code>	<code>"a b"</code>
lstrip	Remove whitespace at left (leading) edge of string	<code>" a b ".lstrip()</code>	<code>"a b "</code>
rstrip	Remove whitespace at right (trailing) edge of string.	<code>" a b ".rstrip()</code>	<code>" a b"</code>
count	Count how many times one string appears in another.	<code>"abracadabra".count("ra")</code>	2
find	Return the index of the first occurrence of one string in another, or -1	<code>"abracadabra".find("ra")</code>	2
		<code>"abracadabra".find("xyz")</code>	-1
replace	Replace occurrences of one string with another.	<code>"abracadabra".replace("ra", "-")</code>	<code>"ab-cadab-"</code>



Notes on String Methods

- These methods don't have to be called on constant strings
 - In fact, they usually aren't

```
>>> element = 'helium'
>>> print element.upper()
HELIUM
>>> print element.replace('el', 'afn')
hafnium
>>> print 'element after calls:', element
element after calls: helium
```
- These methods create new strings
 - They cannot change the strings they're called on because strings are immutable



Chaining Method Calls

- Method calls can be *chained* together
 - If the result of one method call is an object, you can immediately call a method on it

```
element = "cesium"  
print ':' + element.upper()[4:7].center(10) + ':'  
: UM :
```

- Use this in moderation
 - Long chains of method calls are hard to read and debug



Testing for Membership

- Use `in` to check whether one string appears in another
 - Simpler than the `find` method
 - But it doesn't tell you *where* the substring occurs

```
>>> print "ant" in "tantalum"
```

```
True
```

```
>>> print "mat" in "tantalum"
```

```
False
```



Lists

- A list is a *mutable sequence of objects*
 - *Mutable* means that, unlike a string, it can be changed in place
 - *Of objects* means that lists can hold anything and everything
 - Think of it as a one-dimensional array, or vector, that automatically resizes itself as needed
- Write lists by putting values in square brackets
 - The empty list is written []
- Index and slice as you would a string
 - As with strings, Python checks bounds when indexing, but truncates when slicing

```
>>> gases = ['He', 'Ne', 'Ar', 'Kr']
```

```
>>> print gases
```

```
['He', 'Ne', 'Ar', 'Kr']
```

```
>>> print gases[0], gases[-1]
```

```
He Kr
```



Modifying Lists

- Assign a new value to a list element using `x[i] = v`

```
>>> gases = ['He', 'Ne', 'Ar', 'Kr']
```

```
>>> print 'before:', gases
```

```
before: ['He', 'Ne', 'Ar', 'Kr']
```

```
>>> gases[0] = 'H'
```

```
>>> gases[-1] = 'Xe'
```

```
>>> print 'after:', gases
```

```
after: ['H', 'Ne', 'Ar', 'Xe']
```

- The slot must already exist

```
>>> gases = ['He', 'Ne', 'Ar', 'Kr']
```

```
>>> print 'before:', gases
```

```
before: ['He', 'Ne', 'Ar', 'Kr']
```

```
>>> gases[10] = 'Ra'
```

```
IndexError: list assignment index out of range
```



Modifying Lists

- Use `append` to add an element to the end of a list

- Grows the list as needed

```
characters = []
print characters
for c in 'aeiou':
    characters.append(c)
    print characters
[]
['a']
['a', 'e']
['a', 'e', 'i']
['a', 'e', 'i', 'o']
['a', 'e', 'i', 'o', 'u']
```



Concatenation

- Adding strings (or lists) creates a new string (or list) with all the content of the originals

```
>>> element = 'carbon'
>>> mass = '14'
>>> print element + '-' + mass
carbon-14
>>> lanthanides = ['Ce', 'Pr', 'Nd']
>>> actinides = ['Th', 'Pa', 'U']
>>> all = lanthanides + actinides
>>> print all
['Ce', 'Pr', 'Nd', 'Th', 'Pa', 'U']
```

- Can't concatenate a string and a list

- But `list(text)` creates a list whose elements are the characters of the string text

```
>>> water = 'H2O'
>>> print 'before conversion:', water
before conversion: H2O
>>> water = list(water)
>>> print 'after conversion:', water
after conversion: ['H', '2', 'O']
```



Deleting List Elements

- **del deletes a list element**

- Shortens the list (which can cause problems if you're looping over it at the time)

```
>>> organics = ['H', 'C', 'O', 'N']
>>> print 'original:', organics
original: ['H', 'C', 'O', 'N']
>>> del organics[2]
>>> print 'after deleting item 2:', organics
after deleting item 2: ['H', 'C', 'N']
>>> del organics[-2:]
>>> print 'after deleting the last two remaining items:', organics
after deleting the last two remaining items: ['H']
```

- **Can delete slices, too**

```
>>> organics = ['H', 'C', 'O', 'N']
>>> print 'original:', organics
original: ['H', 'C', 'O', 'N']
>>> del organics[1:-1]
>>> print 'after deleting the middle:', organics
after deleting the middle: ['H', 'N']
```

- **del is a statement, not an operator**

- Doesn't "return" the modified list



List Methods

- Like strings, lists are objects, and have methods
- In the examples below, metals is initially ['gold', 'iron', 'lead', 'gold']

Method	Purpose	Example	Result
append	Add to the end of the list	metals.append('tin')	['gold', 'iron', 'lead', 'gold', 'tin']
count	Count how many times something appears in the list	metals.count('gold')	2
find	Find the first occurrence of something in the list.	metals.find('iron')	1
		metals.find('sulfur')	-1
insert	Insert something into the list.	metals.insert(2, 'silver')	['gold', 'iron', 'silver', 'lead', 'gold']
remove	Remove the first occurrence of something from the list	metals.remove('gold')	['iron', 'lead', 'gold']
reverse	Reverse the list in place.	metals.reverse()	['gold', 'lead', 'iron', 'gold']
sort	Sort the list in place.	metals.sort()	['gold', 'gold', 'iron', 'lead']



Notes on List Methodss

- index reports an error if the item can't be found
- reverse and sort change the list, and return None
 - The object equivalent of zero
 - Like 0 and the empty string, it is equivalent to False
- `x = x.reverse()` is a common error
 - It reverses `x`, but then sets `x` to `None`, so all data is lost



For Loops

- Python's for loops over the *content* of a collection (such as a string or list)

- for c in some_string assigns c each character of some_string
- for v in some_list assigns v each value of some_list

```
for c in 'lead':  
    print '/' + c + '/',  
print  
/l/ /e/ /a/ /d/
```

```
for v in ['he', 'ar', 'ne', 'kr']:  
    print v.capitalize()  
He Ar Ne Kr
```

- Of course, you can use any name you like for the loop index variable
- Why? Because it's usually what you want to do



Ranges

- The built-in function `range` creates the list `[start, start+1, ..., end-1]`
 - `end-1` to be consistent with `x[start:end]`

```
>>> print 'up to 5:', range(5)
up to 5: [0, 1, 2, 3, 4]
>>> print '2 to 5:', range(2, 5)
2 to 5: [2, 3, 4]
>>> print '2 to 10 by 2:', range(2, 10, 2)
2 to 10 by 2: [2, 4, 6, 8]
10 to 2: []
>>> print '10 to 2:', range(10, 2)
>>> print '10 to 2 by -2:', range(10, 2, -2)
10 to 2 by -2: [10, 8, 6, 4]
```
- Note the special cases `range(end)` and `range(start, end, step)`
- Note also that `range` may generate an empty list



Ranged Loops

- To loop from 0 to N-1, use `for i in range(N)`
- To loop over the indices of a list or string, use `for i in range(len(sequence))`

```
element = 'sulfur'
```

```
for i in range(len(element)):
```

```
    print i, element[i]
```

```
0 s
```

```
1 u
```

```
2 l
```

```
3 f
```

```
4 u
```

```
5 r
```

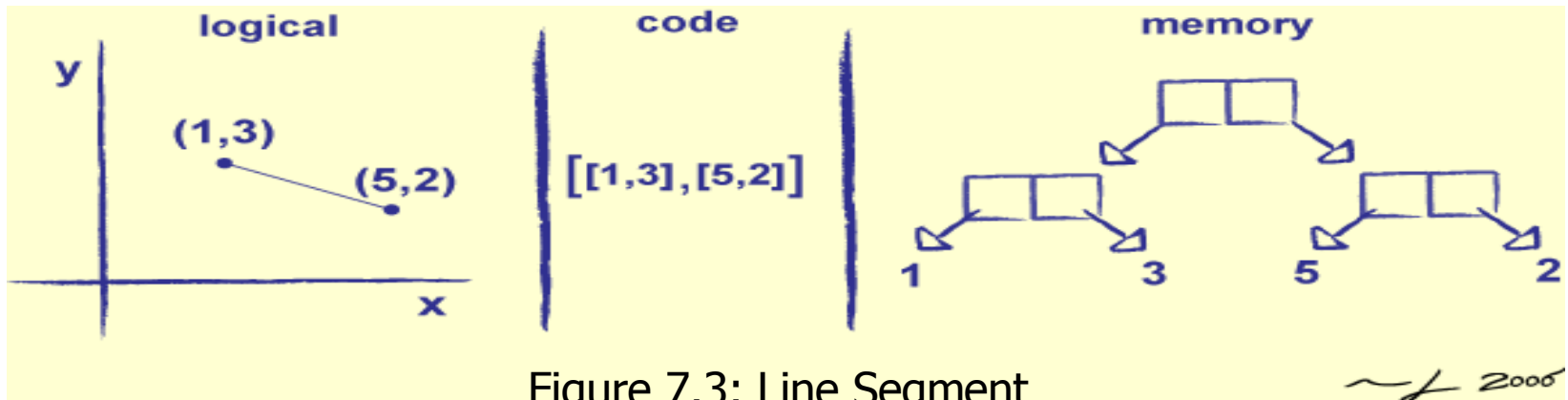


Membership

- `x in c` works element-by-element on lists
 - So `3 in [1, 2, 3, 4]` is `True`
 - But `[2, 3] in [1, 2, 3, 4]` is `False`

Nesting Lists

- Lists can contain other lists
 - E.g., use a list containing two lists to represent a line
 - Indexing from left to right selects elements from the outside in
- ```
>>> elements = [['H', 'Li', 'Na'], ['F', 'Cl']]
>>> print 'first item in outer list:', elements[0]
first item in outer list: ['H', 'Li', 'Na']
>>> print 'second item of second sublist:', elements[1][1]
second item of second sublist: Cl
```





# Aliasing

---

- Nested lists are objects in their own right
  - The outer list stores a reference to the inner list
  - But the inner list does *not* know that it's being referred to
- Subscribing the outer list creates an *alias* for the inner list
  - Another name for the same data
- Changes made through either reference update the same data

```
elements = [['H', 'Li'], ['F', 'Cl']]
gases = elements[1]
print 'before'
print 'elements:', elements
print 'gases:', gases
gases[1] = 'Br'
print 'after'
print 'elements:', elements
before
elements: [['H', 'Li'], ['F', 'Cl']]
gases: ['F', 'Cl']
after
elements: [['H', 'Li'], ['F', 'Br']]
```

# Aliasing in Action

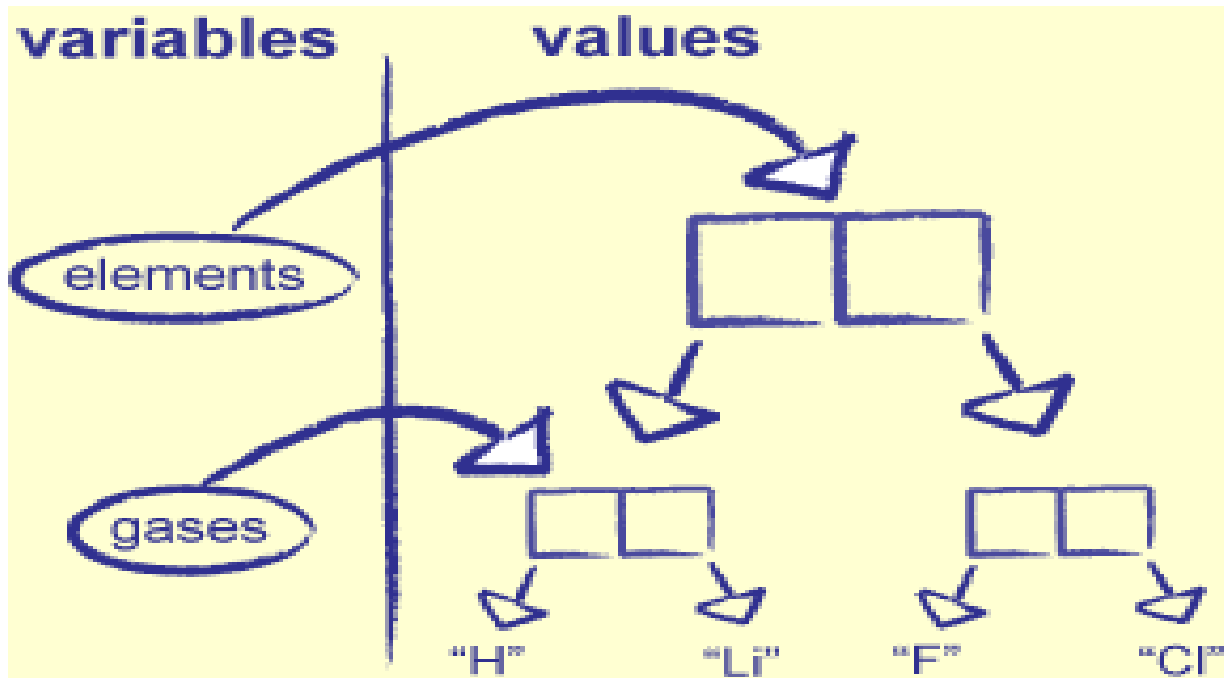


Figure 7.4: Aliasing in Action *2000*



# Indexing vs. Slicing

---

- Indexing and slicing return different types of things for lists
  - Indexing a list returns a reference to the list element
  - Slicing returns a new list containing the selected elements of the original list
- Changes to a slice do *not* affect the original list

```
>>> metals = ['Cr', 'Mn', 'Fe', 'Co', 'Ni', 'Cu', 'Zn']
>>> middle = metals[2:-2]
>>> print 'before'
>>> print 'metals:', metals
>>> print 'middle:', middle
before
metals: ['Cr', 'Mn', 'Fe', 'Co', 'Ni', 'Cu', 'Zn']
middle: ['Fe', 'Co', 'Ni']
```

```
>>> middle[0] = 'Al'
>>> del middle[1]
>>> print 'after'
>>> print 'metals:', metals
>>> print 'middle:', middle
after
metals: ['Cr', 'Mn', 'Fe', 'Co', 'Ni', 'Cu', 'Zn']
middle: ['Al', 'Ni']
```

# Indexing vs. Slicing

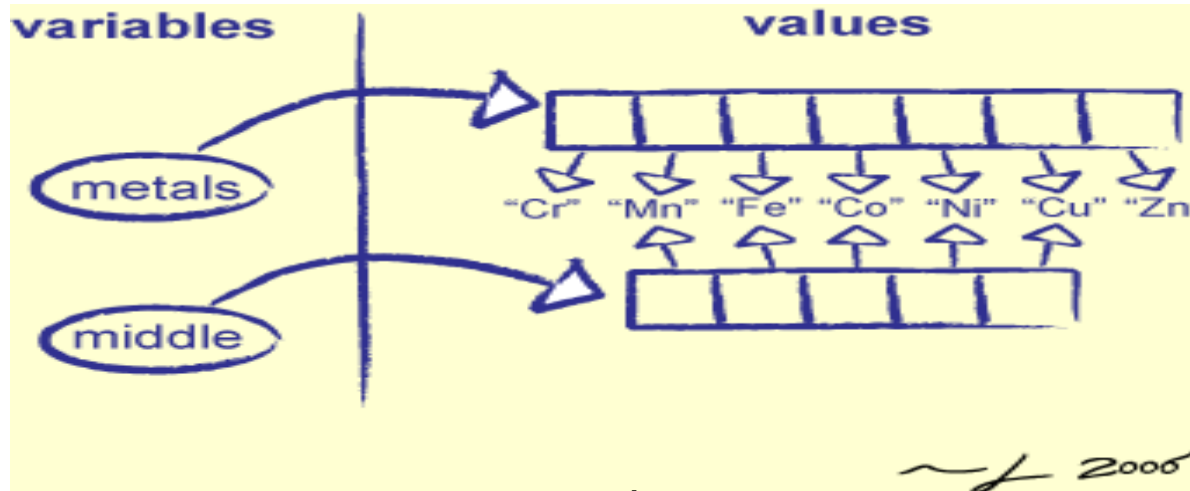


Figure 7.5: Slicing Lists

- Note that copying only goes one level deep
- Don't have to worry about this with strings, since they're immutable
- Draw pictures when you have to
  - And if your pictures are complicated, simplify your code



# Tuples

---

- Python has a second type of list, called a *tuple*
  - Just like a normal list, but immutable (i.e., can't be changed after creation)
  - Written using parentheses instead of square brackets: (1, 2, 3) instead of [1, 2, 3]
  - Empty tuple is ()
  - Tuple with one element must be written with a comma, as in (55,)
    - Because (55) has to be just the integer 55, or the mathematicians will get upset
- Why? Because there are times when Python needs to know that a sequences values aren't going to change
  - We'll meet them *later*
- One of Python's few warts...



# Multi-Valued Assignment

---

- Don't actually need the parentheses around a tuple
  - 1, 2, 3 is the same as (1, 2, 3)
- Allows *multi-valued assignment*
  - left, right = "gold", "lead" assigns "gold" to left, and "lead" to right
- Python converts lists to tuples when necessary
  - left, middle, right = ["gold", "iron", "lead"] works
  - left, right = ["gold", "iron", "lead"] doesn't
    - Number of targets on the left must match the number of values on the right
- Often used to exchange values
  - left, right = right, left does a safe swap



# Unpacking Structures in Loops

---

- Use multi-valued assignment in for loops to unpack structures on the fly

```
elements = [['H', 'hydrogen', 1.008], ['He', 'helium',
4.003], ['Li', 'lithium', 6.941], ['Be', 'beryllium',
9.012]]
```

```
for (symbol, name, weight) in elements:
 print name + ' (' + symbol + '): ' + str(weight)
```

```
hydrogen (H): 1.008
```

```
helium (He): 4.003
```

```
lithium (Li): 6.941
```

```
beryllium (Be): 9.012
```

- Two of the reasons nimble languages are productive are that they:
  - Let you write complex data structures directly
  - Let you take them apart easily



# Files

---

- Use the built-in function `open` to open a file
  - First argument is path
  - Second is "r" (for read) or "w" for write

```
input_file = open('count_bytes.py', 'r')
content = input_file.read()
input_file.close()
print len(content), 'bytes in file'
121 bytes in file
```
- Result is a file object with methods for input and output



# Files

---

| Method     | Purpose                                                                                    | Example                                                 |
|------------|--------------------------------------------------------------------------------------------|---------------------------------------------------------|
| close      | Close the file; no more reading or writing is allowed                                      | <code>input_file.close()</code>                         |
| read       | Read N bytes from the file, returning the empty string if the file is empty.               | <code>next_block = input_file.read(1024)</code>         |
|            | If N is not given, read the rest of the file.                                              | <code>rest = input_file.read()</code>                   |
| readline   | Read the next line of text from the file, returning the empty string if the file is empty. | <code>line = input_file.readline()</code>               |
| readlines  | Return the remaining lines in the file as a list, or an empty list at the end of the file. | <code>rest = input_file.readlines()</code>              |
| write      | Write a string to a file.                                                                  | <code>output_file.write("Element 8:<br/>Oxygen")</code> |
|            |                                                                                            | write does <b>NOT</b> automatically append a newline    |
| writelines | Write each string in a list to a file (without appending newlines).                        | <code>output_file.writelines(["H", "He", "Li"])</code>  |



# Copying a File

---

```
input_file = open('file.txt', 'r')
output_file = open('copy.txt', 'w')
line = input_file.readline()
while line:
 output_file.write(line)
 line = input_file.readline()
input_file.close()
output_file.close()
```

- First statement opens file.txt for reading, and assigns the file object to input\_file
- Second statement opens copy.txt for writing, and assigns the file object to output\_file
- Program then tries to read a line from input\_file
  - If the file is empty, line is assigned the empty string
- As long as there are lines in the input, the program:
  - Writes the most recent line to output\_file
  - Reads the next line
- After the input file is exhausted, the program closes the files
  - Python will close the files automatically when the program exits...
  - ...but it's good practice to tidy up your toys when you're done playing with them



# Looping Over Files

---

- Looping over an input file returns lines of text

```
input_file = open('count_lines.py', 'r')
```

```
count = 0 for line in input_file:
```

```
 count += 1
```

```
input_file.close()
```

```
print count, 'lines in file'
```

```
6 lines in file
```

- Including the terminating newline (or carriage return + newline on Windows)
- Meaningless to loop over an output file



# Other Ways to Copy Files

---

- Could also use this:

```
input_file = open('file.txt', 'r')
lines = input_file.readlines()
input_file.close()
output_file = open('copy.txt', 'w')
output_file.writelines(lines)
output_file.close()
```

- Read all the lines from the input file into a list
- Write the lines in that list to the output file



# Other Ways to Copy Files

---

- Or this:

```
input_file = open('file.txt', 'r')
output_file = open('copy.txt', 'w')
for line in input_file:
 line = line.rstrip()
 print >> output_file, line
input_file.close()
output_file.close()
```

- `print >> file` sends `print`'s output to a file
- Remember that it automatically appends an end-of-line marker
  - Which is why the program above strips whitespace off the end of the string before printing it



# Summary

---

- The basic features of most modern programming languages are the same
  - Strings, lists, file I/O, ...
- Only issue is how they're presented
  - Python's syntax is clean and consistent
- You'll soon start to wonder why other languages still rely on curly braces...