Lecture 4

- Good Software Practices (chapter 7)
- Section 2.2: lists (review)
- Section 2.3: Other Sequence Classes:
  `str` and `tuple` classes
- Section 2.4: Numeric Types: `int`, `long`, and `float`
- Section 2.5: Type Conversions

Good Software Practices

The program:

```
def main():
   print("This program finds ....") # explain to the user
   # what does this program do

   a,b = eval(input("Enter two integers, separated by
comma: "))

   # echoing the data to the screen
   print("You entered: ", a, " and", b)

  guess = min(a,b) # taking smallest of a and b

   while (a%guess != 0) or (b%guess != 0):
      guess -= 1
      print("The new guess is ", guess) #show guess
change


   print("ANSWER: The gcd of", a, "and", b, "is", guess)

main()
```

Good Software Practices

Introduction

Many beginning programmers are tempted to sit down and start writing code.

This, however, is not a very good strategy. Poorly designed code is much harder to maintain and sometimes must be scrapped entirely.

A well written program generally stems from a good initial design.

Good Software Practices

Before we write any portion of a large program we should consider its overall design.

One of the challenges: to *identify potential classes*, and the way in which *objects from these classes interact*.

Good Software Practices

Starting point: envision the use of our final product and design a *top-level class and its interface*.

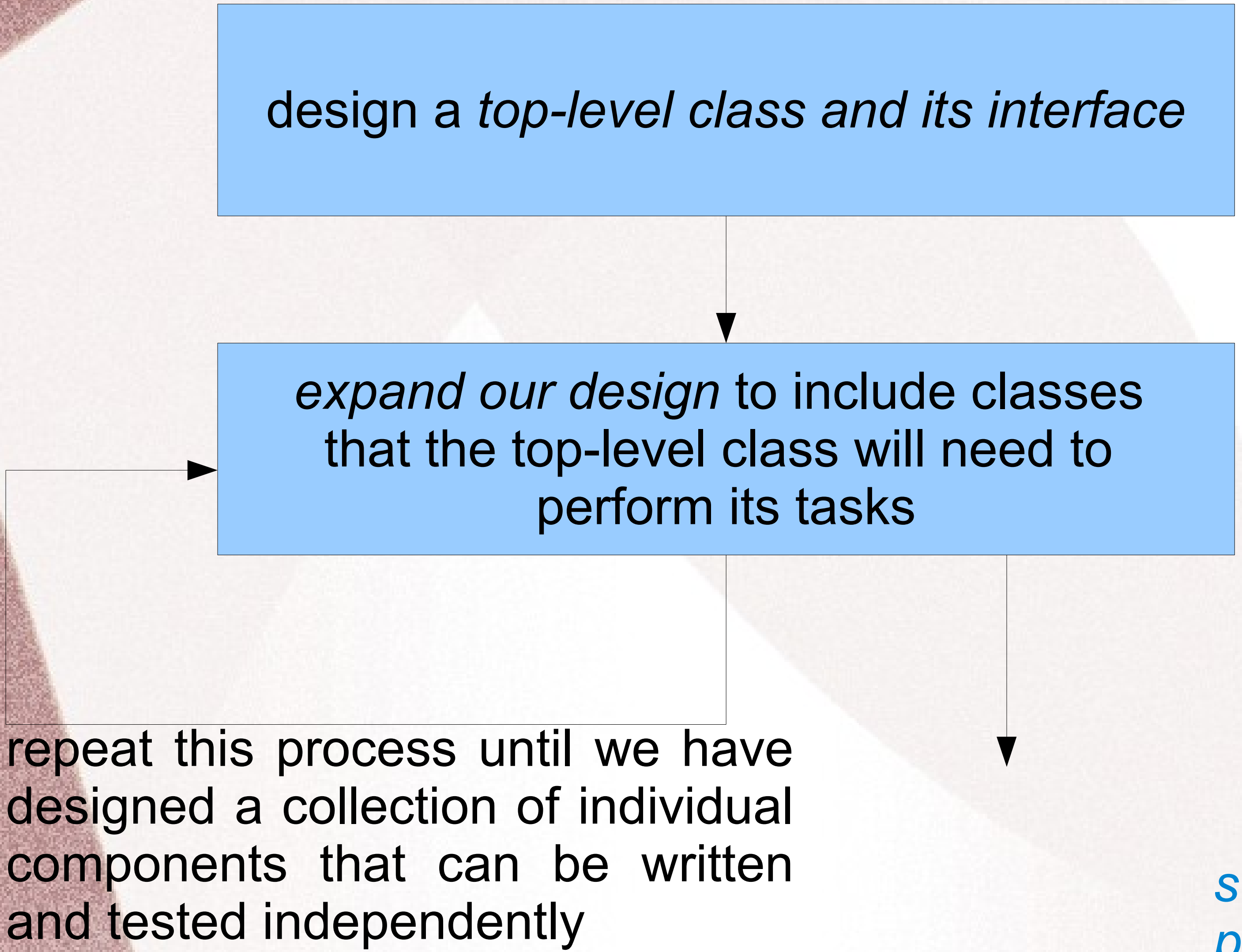
- from there we expand our design to include classes that the top-level class will need to perform its tasks.

This process is repeated until we have designed a collection of individual components that can be written and tested independently.

- *top-down design*

Good Software Practices

Top-Down Design



design a *top-level class and its interface*

expand our design to include classes that the top-level class will need to perform its tasks

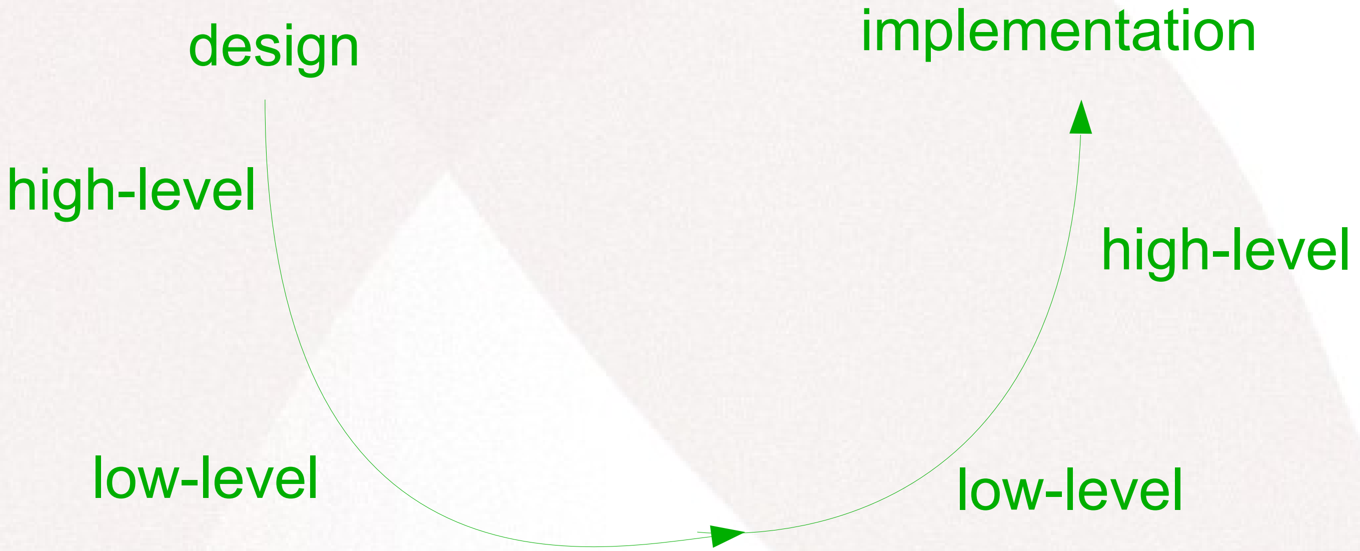
repeat this process until we have designed a collection of individual components that can be written and tested independently

summary of the previous slide

Good Software Practices

Once the design is fixed, implementation should start at the lowest level and work up to the top level.
- this technique is known as *bottom-up implementation*

Top-Down Design and Bottom-Up Implementation



For the the approach to be successful, every component must be tested thoroughly before moving on to the next higher level, as the higher-level code will depend upon use of the lower-level components.

Good Software Practices

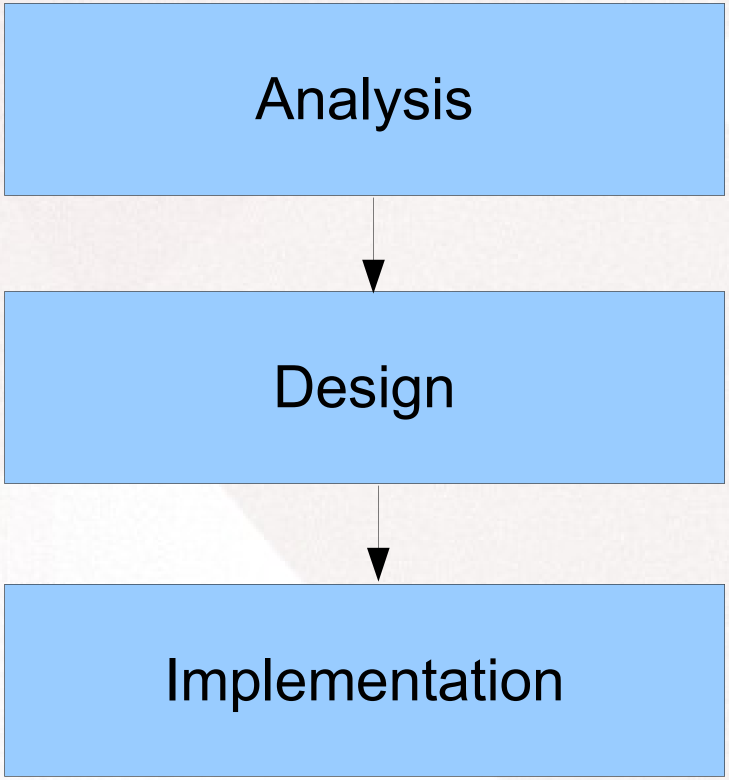
Top-Down Design and Bottom-Up Implementation

Time spent on top-down design is rewarded in the form of:

- better design, that is easier and quicker to implement in a long run
- less likely to write a code that later will be thrown away due to unforeseen complications
- *modularity* (the existence of smaller independent pieces):
  - pieces can be implemented and tested in parallel by team of software developers
  - code reuse across multiple portions of a program in other projects (since some of the components might have more general values)

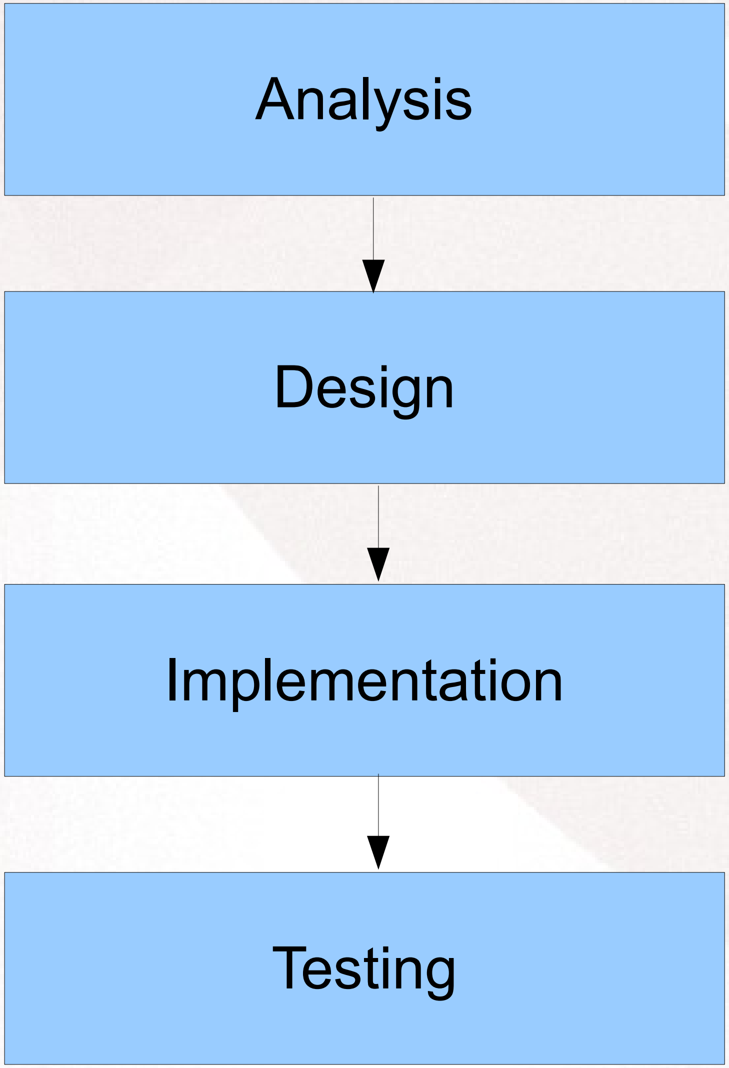
Good Software Practices

Object Oriented Development



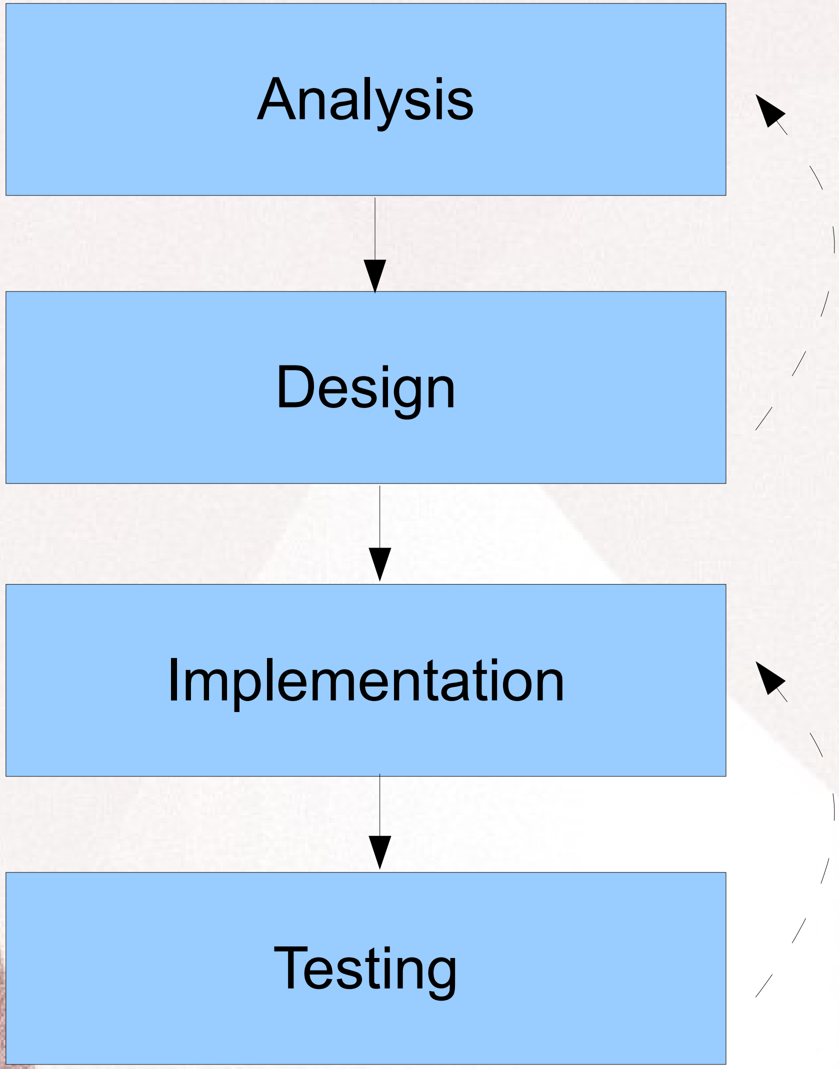
Good Software Practices

Object Oriented Development



Good Software Practices

Object Oriented Development



gathering requirements

see top-down design

see bottom-up implementation

Good Software Practices

Formal Documentation

As you write more complicated code, the importance of good documentation increases.

Documentation informs another programmer how to properly use your classes and functions, and serves as a formal specification of the promised behavior.

From a planning perspective, the expected behavior of a class or a function should be well defined and documented before the actual implementation is written.

# - symbol for directly embedding commends within source code (ignored by interpreter, yet visible for programmer)

Python supports another style of documentation – using *docstrings*

Good Software Practices

Formal Documentation

Python supports another style of documentation – using *docstrings*

- these strings are:

- visible in the source code,
- can be seen in the Python interpreter through the use of the *help* command, and
- can be used to generate documentation on web pages using a corresponding utility *pydoc*

When this style of documentation is useful:

when a programmer wishes to use a class or function written by another, without taking time to closely examine the original source code

for example, Python's built-in classes or functions.

Good Software Practices

Formal Documentation

A *docstring* is technically a string literal that occurs as the very first element within the body of a class or function.

Typically, triple quote delimeters (" " ") are used since these allow for multiple strings.

A docstring should begin with a brief one line description.

If further explanation is warranted, such as the purpose or type of parameters and return value, that information should be provided after a subsequent blank line within the docstring.

see programs gcd-alg2.py and gcd-alg3.py

2.2 Using Objects: the `list` class

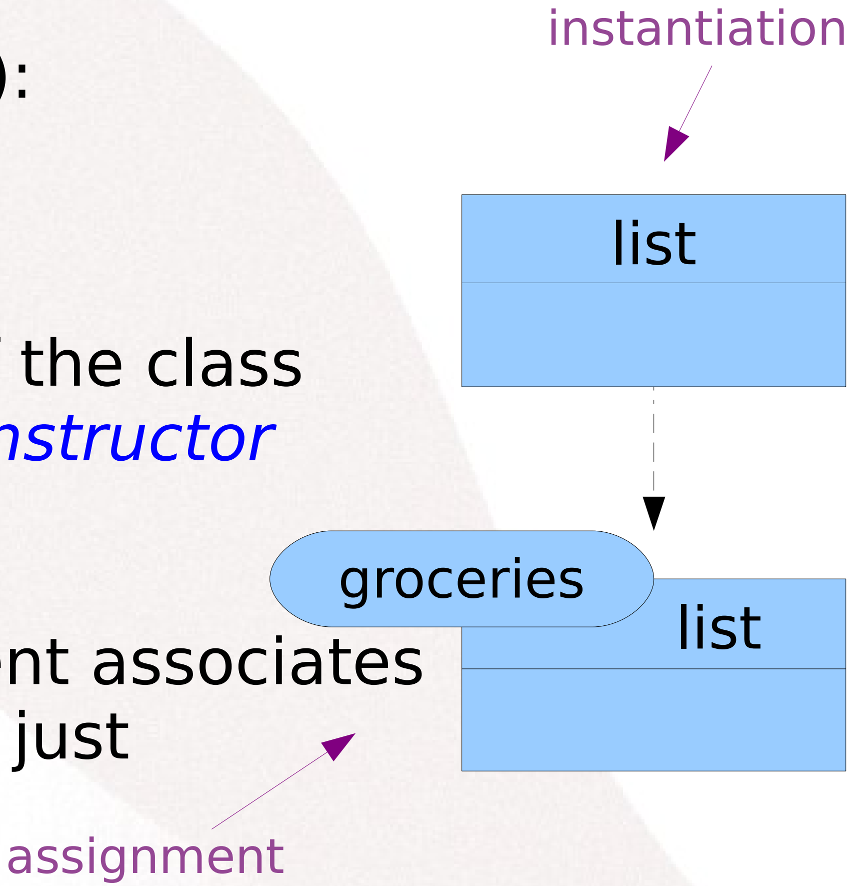
- a built-in class
- used to maintain an ordered list of items

Example(REVIEW of terminology):
Let's make up a list of groceries:

```
groceries = list()
```

-`list()` creates an instance of the class `list` (by invoking the *constructor* of the class)

- then the assignment statement associates the identifier `groceries` with just created object



2.2 Using Objects: the *list* class

```
def main():
    groceries = list()

    groceries.append('milk')
    groceries.append("bread")
    groceries.append("cream cheese")

    groceries.insert(0,"sour cream")

    groceries.remove("bread")

    groceries.sort()

    print len(groceries)

main()
```

adding elements to the end of the list

adding 'sour cream' to the top of the list

removing the first occurrence of 'bread' from the list

sort list in alphabetical order

print the current lenth of the list

See more methods (functions) on page 42

See an example of a program in
list-example1.py

Keep in mind:
Objects/instances of *list class are mutable*, i.e. can be changed/updated after creation/ construction.

Other Sequence Classes: *str* class (review)

- a built-in class
- immutable objects
 (once constructed, can no longer be modified)
- used to represent ordered sequences

Construction of strings:

`str()` - invocation (call) of strings' constructor
(returns empty string '')

or

`phrase = “Hello! Long time no see.”`
(using literal form)

See pages 56-57 for the list of behaviors of strings:
- return the information about existing string
- generate a new string as a result
- convert between strings and lists of strings

Examples of work with strings:

```
string = input("Please, input any sentence:")
```

```
string.count('a')
```

– counts the number of occurrences of letter 'a' in the string

`len(string)` – finds the length of the string

```
print(string.upper())
```

– convert all letters to upper case, print

`print(string)` – the value of “string” is unchanged

```
position = string.find(' ')
```

– finds the first occurrence of whitespace

Examples of work with strings:

```
str2 = string[:position]
```

– assigns the head of the string (before the whitespace) to “str2”

```
pos2=string.find(' ',position+1)
```

– finds the second occurrence of whitespace

```
str3=string[position+1:pos2]
```

– assigns the content of the string in between the first and the second whitespace to “str3”

```
print(str2 + " ho-ho-ho " + str3)
```

– prints the new strings with “ho-ho-ho” in between

```
print(string * 3)
```

– prints three times concatenated string

Examples of work with strings:

See string_example.py

Did you notice that the method `len` is supported in both list and string classes?

Examples of work with strings:

See string_example.py

Did you notice that the method `len` is supported in both list and string classes?
- *Polymorphism*

Examples of work with strings:

More uses:

```
string=”Hello, Buy, 123, Thank you”
print(string, type(string))
l = string.split(',')
print(l, type(l))
```

See string_example2.py

Other Sequence Classes: `tuple` class

- a built-in class
- immutable objects
 (once constructed, can no longer be modified)
- used to encapsulate multiple pieces of information into a single composite object that can be stored or transmitted.

Construction of lists (using literal form)**:**

```
contact_info = (718-987-2345, 917-765-2314, AAA@mail.com)
```

or

```
white = (255, 255, 255)
```

Tuples support all the nonmutating behaviors of lists (see page 42)

Numeric Types: *int, long, and float*

- are built-in classes
- immutable objects

- int class is the most common of the numeric types; has a maximum magnitude for a value stored (depends on the computer architecture)

- long class is used to represent integers with arbitrarily large magnitudes

- Python automatically converts from the use of int to long, when necessary

- float class is used for representation of irrational numbers (that are stored in approximate form), for example $\sqrt{2}$, and also to store decimal numbers.

Numeric Types: *int, long, and float*

All three classes support a common set of operations (see page 59) - another example of *polymorphism*

Recall tricky cases:

result = 5/2

- 'result' will have value 2 assigned to it in Python 2
- this was “corrected” in Python 3,
  i.e. 5/2 will produce 2.5,
  but it is good to keep this case in mind.

Numeric Types: *int, long, and float*

Type conversions (casting):

```
n = int(2.8)
```

– 'n' will have value 2 assigned to it (truncation)

```
n = round(2.8)
```

– 'n' will have value 3 assigned to it

```
n = round(3.141592654,4)
```

– 'n' will be 3.1416

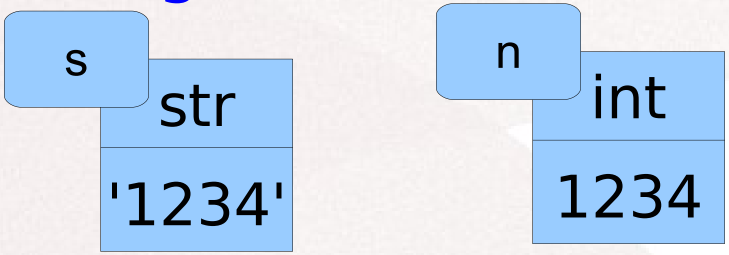
```
n = float(15)/4
```

– 'n' will be assigned 3.75

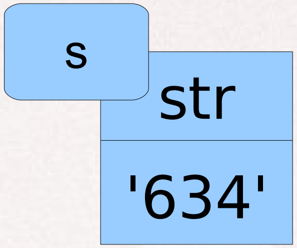
Numeric Types: *int, long, and float*

Type conversions (casting) continues:

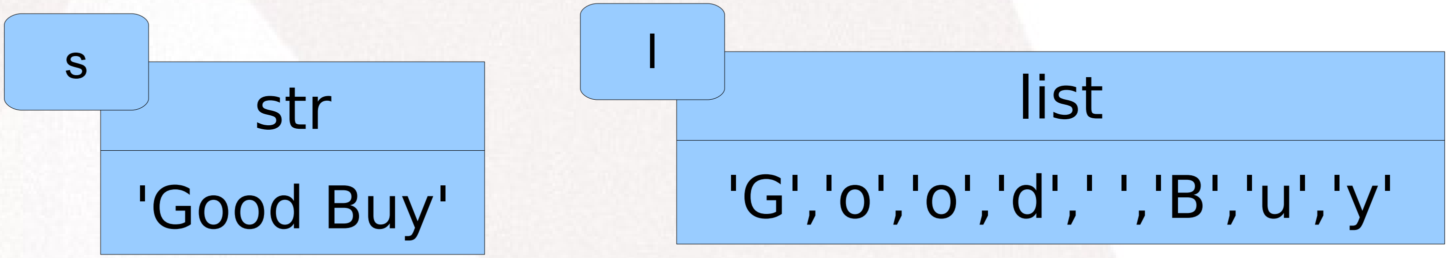
s='1234'
n=int(str)


s
str
'1234'
n
int
1234

s=str(634)


s
str
'634'

s=”Good Buy”
l=list(s)


s
str
'Good Buy'
l
list
'G','o','o','d',' ','B','u','y'

Low-level encoding of characters (ASCII, or Unicode):

`ord('a')`   `chr(97)`

`Visit page http://www.asciitable.com/ for ASCII codes Table.`

In-Class Work

1. Assume that person is a string of the form 'firstName lastName'. Give a command or series of commands that results in the corresponding string 'lastName, firstName'.

2. Consider the following code fragment:

```
message = 'Hello, my name is Frank'
start = message[:18]
name = message[18:]

letters = list(name)
letters.remove('F')
letters.sort()
letters[1]='N'

name = ''.join(letters)
name = name.replace('r','.')

print(start+name.capitalize())
```

What is the output of the print statement?

Homework Assignment

1. page 84 / 2.24

2. Write a program that takes a *string of words separated by a space* from the user; converts is to a list of strings (use whitespace as the separator); then if an element of the list is a string of digits, the program converts it to a whole number (positive integers), and replaces the corresponding element of the list with this number. The conversion should be done for all such elements (that are whole numbers, written as a string).

Example of input-output:

Input (from the user): Hello buy mind 5 abc 12 0 1243 thank you sorry 432 mind 5 Hello George

Output: ['Hello','buy','mind',5,'abc',12,0,1243,'thank you','sorry', 432,'mind',5,'Hello','George']

Your program must be well commented. Otherwise 0.5 points will be taken off.

next slide

Homework Assignment (continues)

3. Write a program that takes a list of “words” as an input (from a user, from the keyboard; “words” are separated by commas), echoes it to the display/screen and then allows the user to remove all occurrences of an item provided by user from the list.

Comments: the user will provide something of this kind as an input: mother,hello,1823,buy,thank you,you are welcome. Your program will take it as a string. In order to generate a list from it you can use method `split`

Example: `listWords=text.split(',')`