- An example of a software development process: the *day-month to day-of-year* problem
- String operations
- File operations
- Exercises on the above

- Robotics: motion commands as an example of the *input-compute-output* pattern

Please sit with your robot partner

- *Well, sit with your robotics partner (presumably your partner is not a robot…)*

Outline

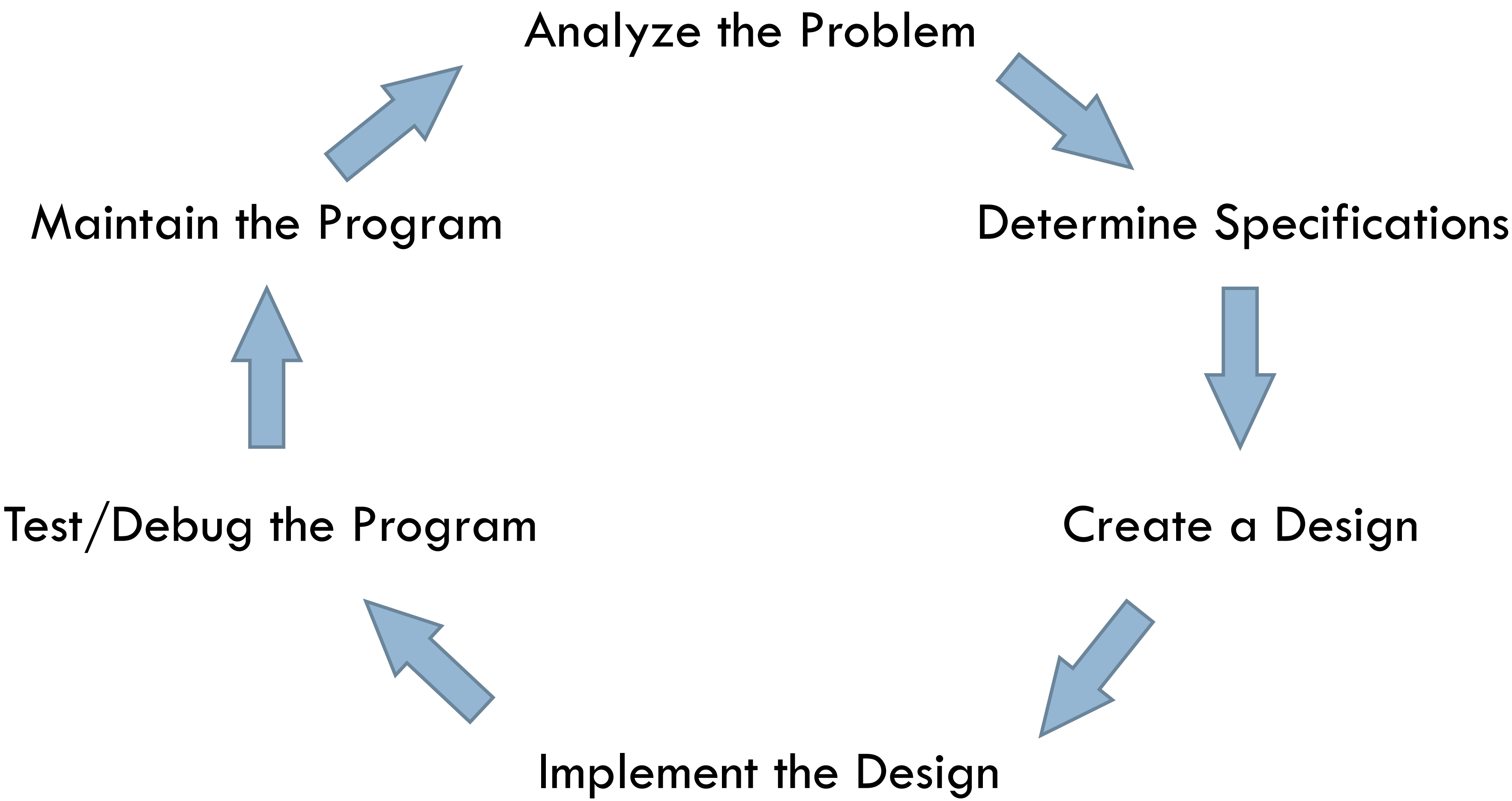
- Lists – review
  - *range* function for creating a List
  - Looping through a List
  - Applying the *sum* function to a List
- An example of a software development process: the *day-month to day-of-year* problem
- Strings
  - How strings are represented: the *ord* and *chr* functions
  - Encodings: ASCII, extended ASCII, Unicode
  - Formatting with the % operator
- *input* versus *raw_input*
- File operations: open, close, read, write

Q1

Day, Month → Day of year

- When calculating the amount of money required to pay off a loan, banks often need to know what the "ordinal value" of a particular date is
  - For example, March 6 is the 65th day of the year (in a non-leap year)
- We need a program to calculate the day of the year when given a particular month and day

The Software Development Process



Phases of Software Development

- **Analyze:** figure out exactly what the problem to be solved is
  - *Need to be able to find the day of the year, when given month and date.*
- **Specify:** WHAT will program do? NOT HOW.
  - *User provides month (three letters, lowercase) and day of month (integer). Program calculates and prints the day of the year. Not required to work for leap years.*
- **Design:** SKETCH how your program will do its work, design the algorithm
  - *Use two parallel lists, one of month names, one of month lengths. Once we get the month and day, use a loop to add up the lengths of the previous months.*
- **Implement**: translate design to computer language
- **Test/debug**: See if it works as expected.
  - bug == error, debug == find and fix errors
- **Maintain**: continue developing in response to needs of users

String Representation

- Computer stores 0s and 1s
  - These 0's and 1's are called ***bits***
  - Numbers are stored as sequences of 0s and 1s
    - Fixed-length sequences for *int* and *float*
    - Arbitrarily long sequences for *long*
  - What about text?
- Text is also stored as sequences of 0s and 1s
  - Each character has a code number
  - Strings are sequences of characters
    - So Strings are stored as sequences of code numbers
  - Does it matter what code numbers we use?
    - No, as long as we're consistent – see next slide

Translating:

- ***ord***(<char>)
- ***chr***(<int>)

```
>>> ord("R")
82
>>> ord("r")
114
>>> chr(114)
'r'
>>> chr(115)
's'
>>> chr(113)
'q`
>>> ord(` `)
32
>>> ord(`$')
36
```

Q2-3

Consistent String Encodings

- Needed to share data between computers
- Examples:
  - ASCII—American Standard Code for Info. Interchange
    - “Ask-ee”
    - Standard US keyboard characters plus “control codes”
    - 8 bits per character
  - Extended ASCII encodings (8 bits)
    - Add various international characters
  - Unicode (16+ bits)
    - Tens of thousands of characters
    - Nearly every written language known

Q4

String Formatting

- The % operator is *overloaded*
  - Multiple meanings depending on types of operands
- What does it mean for numbers?
  - Answer: *remainder*
- Other meaning for `<string> % <tuple>`
  - Plug values from *tuple* into “slots” in *string*
  - Slots given by *format specifiers*
  - Each format specifier begins with % and ends with a letter
  - Length of *tuple* must match number of slots in the *string*

Format Specifiers

- Syntax:
  - **%<width>.<precision><typeChar>**
- ***Width*** gives total spaces to use
  - 0 (or width omitted) means as many as needed
  - 0*n* means pad with leading 0s to *n* **total** spaces
    - *n* without the zero means pad with leading spaces instead of zeroes
  - -*n* means "left justify" in the *n* spaces
- ***Precision*** gives digits after decimal point, rounding if needed.
- ***TypeChar*** is:
  - f for float, s for string, or d for decimal(i.e., int)
- Note: this ***returns*** a string that we can print
  - Or write to a file using **`write(string)`**, as you'll do on today's homework

However, rounding can be flaky due to underlying base 2:

```
>>> print '%.2f' % 2.375
2.38
>>> print '%.2f' % 2.385
2.38
```

A typical use of formatting specifiers is to produce tabular output. See your homework for an example.

Q5

input() and *raw_input*() are related through the *eval* function

- *input*(…) – evaluates the input and returns the result of the evaluation
  - User enters 4.56 → floating point number is returned
  - User enters 68 → integer number is returned
  - User enters "88" → string "88" is returned
  - User enters x → value of variable *x* is returned (error if *x* is not defined)
- **raw_input**(…) – returns the input as a string
  - User enters 4.56 → string "4.56" is returned
  - User enters x → string "x" is returned
- *eval*(<string>) – evaluates the given string as if it were a Python expression
  - ```
    x,y = 7,5
    eval("3 + 4")  →  7
    eval("x * y")  →  35
    ```

> `input(…)`
> is equivalent to
> `eval(raw_input(…))`

Q6-7
    ```

File Processing

- Manipulating data stored on disk
- Key steps:
  - *Open* file
    - For reading or writing
    - Associates file on disk with a *file variable* in program
  - *Manipulate* file with operations on the file variable
    - *Read* or *write* information
  - *Close* file
    - Causes final “bookkeeping” to happen

Note: disks are slow, so changes to the file are often kept in a ***buffer*** in memory until we close the file or otherwise “flush” the buffer.

Q8

File Writing in Python

- Open file:
  - Syntax: <filevar> = ***open***(<name>, <mode>)
  - Example: `outFile = open('average.txt', 'w')`
    - Replaces contents!
- Write to file:
  - Syntax: <filevar>.***write***(<string>)
  - Example: `outFile.write("And this isn't my nose.\ It's a false one.")`
- Close file:
  - Syntax: <filevar>.***close***()
  - Example: `outFile.close()`

File Reading in Python

- Open file: `inFile = open('grades.txt', 'r')`
- Read file:
  - <filevar>.***read***() Returns one **BIG** string
  - <filevar>.***readline***() Returns next line, including \n
  - <filevar>.***readlines***() Returns **BIG** list of strings, 1 per line
  - ***for <lineVar> in <filevar>*** Iterates over lines efficiently
- Close file: `inFile.close()`
- Exercise: write a program called filePractice.py that:
  - Asks the user for 3 phrases and writes the 3 phrases onto file "output.txt", each phrase on its own line
  - Asks the user for a filename and then prints all the lines of the file 4 times, once for each of the 4 read-styles listed above

A "Big" Difference

- Consider:
  - ```
    inFile = open ('grades.txt', 'r`)
    for line in inFile.readlines():
      # process line
    inFile.close()
    ```
  - ```
    inFile = open ('grades.txt', 'r`)
    for line in inFile:
      # process line
    inFile.close()
    ```
- Which takes the least memory?
  - Answer: the second approach, because in it Python reads lines into memory one at a time and only as needed instead of all at once, as in the first approach

Q9

- Write a program called goTest.py that:
  - Asks the user for a distance, in inches
  - Goes that many inches using the *go* function and an appropriate *sleep*
  - Prints how many inches the robot thinks it went using

    robot.getSensor("DISTANCE")
  - Repeats the above using the go function and an appropriate *waitDistance*
- *In both cases, also measure (with a ruler) how many inches the robot went. See how accurate your robot is for 3 inches, 6 inches, 24 inches.*

Practice

- Hand in quiz
- Do the
- Start working on HW5
- On Angel
  - Lessons → Homework → Homework 5 → Homework 5 Instructions