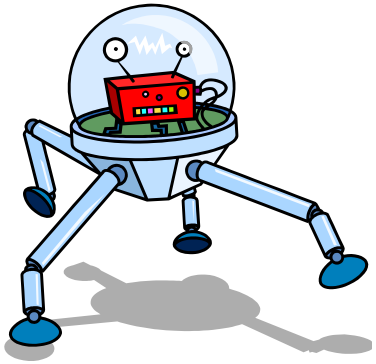
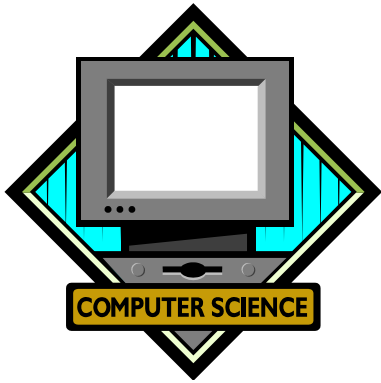
Caltech/LEAD Summer 2012 Computer Science



Lecture 8: July 17, 2012



How to Think About Programming

Programming: The Challenge

- Programming can be very difficult if you aren't used to it
- Problem: computers are very simple
  - Very limited set of operations
  - Even high-level languages like Python have relatively simple sets of operations to perform
  - Very rigid rules about what data is visible to a piece of code, and when/how it can be manipulated

Programming: The Challenge (2)

- Two key aspects of becoming a good programmer

- Learn computational model as well as possible!
  - The better you understand how the language works, the easier it is to understand what's going on
- This is the basic approach of science in general:
  - Understand the fundamental principles and behaviors of your environment…
  - …then exploit these underlying principles to solve problems we care about

Computational Model?

- By "computational model" we mean how the computer and programming language behave
  - How we can manipulate data structures
  - What kinds of operations are acceptable, and how they behave
- For example:
  - Variables declared inside a function are not visible outside the function
  - Strings and tuples are immutable
  - Lists are mutable
  - Dividing two integers produces an integer
  - …and so forth

Programming: The Challenge (3)

- Must also learn to translate a high-level problem into the program that solves it
- Remember: computers are very simple
  - Almost never a single function or operation that will solve your problem instantly
- Must *decompose* problem into a sequence of operations the computer can understand
  - Critical to understand computational model when you do this!
  - Instructions must pass data to each other through variables, lists, etc.

Decomposing a Problem

- So far, assignments have decomposed the programming tasks for you
  - (And, for the most part, we will continue to)
- When working on your own projects, you must learn to decompose projects yourself
- On one side: the task you want to complete
  - e.g. compute a score for a blackjack hand
- On other side: what computer can actually do
  - e.g. basic data structures, built-in functions like `sum()` and `print`, modules like `random`, etc.
- **Your program needs to bridge this gap.**

Technique: Pseudocode

- Just like writing a paper, it helps to outline what the program or function must do
  - Helps you deconstruct the problem into its component steps, so you can bridge that gap

- Example: tell me how to make a sandwich
  - Things: bread, cheese, lettuce, tomato, knife
  - Places: counter, refrigerator, cabinet
  - Actions: go to [place], pick up [thing], put down [thing], open [place] door, slice [thing], etc.

Technique: Pseudocode (2)

- Outline of a computer program is often called *pseudocode*
  - A mix of code-like constructs (e.g. `for`, `if`, `while`, variables, etc.) and normal-language descriptions
  - There's no formal specification of pseudocode: computers don't execute it! It's for humans.
- A great place to put your pseudocode: in the bodies of your functions, before you write them
  - Create the function signature, write a docstring
  - In the body of the function, write pseudocode as comments, outlining what the function must do
  - Then, just translate pseudocode into actual code
  - (Easy in Python – it's already so similar to English!)

Technique: Pseudocode (3)

- Outlining your program will tell you if you are actually ready to write it
- If you can't write very detailed pseudocode:
  - You probably don't understand how to solve the problem yet
  - Go back and study the problem in more detail, then devise a solution.
- You might also not understand how the solution maps to the language or modules
  - (may need to research what language/module facilities are available that could help you out)

General Approach

- Programs always have a specific function as the entry-point
  - This is where the program starts running when you invoke it
- Create this entry-point
  - Write a docstring describing the overall purpose of the program
- In the function's body, write pseudocode that describes what the program will do

General Approach (2)

- For simple programs, can often implement the entire program in one function
- More complex programs almost always require many functions
- The pseudocode will help you identify what other functions you need to create
  - e.g. if some step in the function outline will clearly require significant code to implement

Decomposing Problems into Functions

- No hard-and-fast rules about when to create a function for a given operation
  - Practice will develop your coding intuition
- Some guidelines:
  - If a computation has a good chance of being generally useful, make it a separate function
    - Yes, even if you only use it in one place so far. Trust me. ☺
  - If a computation requires significant code and/or local variables to implement, it should probably be in its own function

Decomposing Problems into Functions

- Corollary: function implementations should never be very long!
  - ideally under 50 lines of code
  - If a function is longer than this, probably needs to be decomposed into smaller functions
  - Identify clean places where the function can be broken apart
    - e.g. independent sub-computations in the function
- This process is called *refactoring*
  - Rearranging the code in your program to improve its structure and maintainability

Decomposing Problems into Functions

- Also: good programmers almost never repeat themselves!
  - If you repeat/copy a chunk of code, even just a few lines, put it into a function!
  - (Similarly, if a chunk of code repeats a particular line of code more than e.g. twice, you probably should be using a loop.)
- Several reasons:
  - If code is repeated in multiple places, and you find a bug in that code, you need to find all the places and fix that code
  - Repeating chunks of code makes your program larger, which often makes it slower to execute

Writing Functions

- When you need a particular operation, and there isn't already a function to do it:
- Just create a function for it!
  - Don't be afraid to dive right in, even if you don't know exactly how it will work
- Come up with a function name and write descriptive comments (*i*.e. a docstring) for it
- Also, give examples of how the function would be called, in your docstring
  - Will help you understand what it needs to do

Writing Functions (2)

- As with the main program, once you know how the function will be used, you can start outlining its operation in pseudocode
  - Initially, pseudocode may simply be a description of the steps to solve the problem
  - Keep refining description until you see exactly how to translate it into real code
- Finally, translate pseudocode into real code
- If you need more functions to implement a function, repeat this process!

Example: Blackjack Hand

- Need a function to score a blackjack hand
  - Write a signature for the function
  - Don't know how it will work yet…
- Come up with a clear name for the function
  - **`blackjack_score`** wouldn't be bad
  - It needs the list of cards in the hand too…
- Write a docstring describing the function:

```
def blackjack_score(cards):
    '''
    Given a list of cards representing a
    blackjack hand, computes a score.
    '''
    pass
```

Example: Blackjack Hand (2)

- Still don't know exactly how it will be used
- Write some examples in the docstring

```
def blackjack_score(cards):
    '''
    Given a list of cards representing a
    blackjack hand, compute a score.

    Examples:
        blackjack_score([]) == 0
        blackjack_score(['Ad', 'Kh']) == 21
        blackjack_score(['Ah','10c','7s']) == 18
    '''
    pass
```

Example: Blackjack Hand (3)

- Could obviously put much more detail in the docstring (and we definitely should)
  - Describe how blackjack hands are scored
    - *i.e.* 2-10 are face-value, J/Q/K are worth 10, Aces are worth 1 or 11
    - Choose value for individual Aces to keep from busting (i.e. total score > 21)
  - Describe how cards are represented:
    - Last character specifies suit: h = hearts, s = spades, d = diamonds, c = clubs
    - Remaining characters specify value: 2-10, J, Q, K, A

Example: Blackjack Hand (4)

- Finally, outline the function's behavior in pseudocode!

```
def blackjack_score(cards):
    ''' ... docstring ... '''

    # Count how many aces are in the hand.
    # Add up the total score of the hand,
    #     using 11 for the value of Aces.
    # If total score > 21, start changing value
    #     of aces from 11 to 1, until
    #     score is <= 21 or we run out of aces.
    # Return total score.
    pass
```

  - At this point, our "pseudocode" is just a description!
  - Still not detailed enough to write code. But, at least we're underway.
  - Keep refining this until we can translate it into code.

Example: Blackjack Hand (5)

- You can continue this example on your own
- We have already learned more detail about how to implement this function:

```
# Count how many aces are in the hand.
# Add up the total score of the hand,
#     using 11 for the value of Aces.
# If total score > 21, start changing value
#     of aces from 11 to 1, until
#     score is <= 21 or we run out of aces.
# Return total score.
```

- Clearly need a variable for total score of hand, plus another variable to count number of Aces…
- Add them into the pseudocode, and refine the outline to look more like real code

Iterative Refinement

- Programming is a process of **iterative refinement**
  - You almost never know the full program when you start writing code (unless it's *really* simple)
- As each part of problem is solved, it shows more clearly how remaining parts will work
  - It's not just "writing a program," it's exploring a problem!
  - Solving the easier parts of the problem sheds light on the harder parts
  - Basically always write programs easy-to-hard ☺

Wishful Thinking

- Also, don't be afraid to use a function before you have defined specifically how it will work!
  - *i.e.* you know you need a function that does X, and you know how you need to invoke it…
  - …but you haven't yet figured out how the function will work
- Don't let this stop you!
  - Create a function signature, document what it will do (when it's finished), and give some example usages
  - Write the code that uses this function…
  - …then come back to the function later and finish it

LEAD
Leadership Education And Development

Programming Patterns

- For remainder of this lecture, review some *really* basic programming patterns
- When you write functions:
  - Write the code for the function using the process outlined earlier
  - Then, review the code to see if it makes sense
- There are specific questions I always ask myself when I write and review my code
  - Best to catch any bugs or issues right away
  - Much less frustrating this way!

Patterns: If-Statements

- If-statements are required when code must handle different values in different ways
- Useful questions when writing if-statements:
  - Does the if-statement actually handle every possible data value that can reach it?
    - Are there possible values that *should be* handled, but *aren't* handled?
    - Are there possible values that you definitely don't want to handle? (If so, document them in your comments…)
  - Can a data value be handled by multiple branches in the if-statement? (Should it? Should it not?)

Patterns: If-Statements (2)

- Example:

```
def compute_interest(balance):
    ''' Computes interest rate on a bank
    account, based on its balance.  Maximum
    allowed balance is $50,000. '''
    if balance <= 5000:
        rate = 1.05
    if balance <= 20000:
        rate = 1.06
    if balance <= 50000:
        rate = 1.07
    return rate
```

- Problems?

Patterns: If-Statements (3)

- This function will return 1.07 for all balances

```
def compute_interest(balance):
    ''' Computes interest rate on a bank
    account, based on its balance.  Maximum
    allowed balance is $50,000. '''
    if balance < 5000:
        rate = 1.05
    if balance < 20000:
        rate = 1.06
    if balance < 50000:
        rate = 1.07
    return rate
```

- Balances < 5000 will trigger multiple if-branches

Patterns: If-Statements (4)

- This is one of several reasons why we strongly encourage the use of **elif** !!!

```
def compute_interest(balance):
    ''' Computes interest rate on a bank
    account, based on its balance.  Maximum
    allowed balance is $50,000. '''
    if balance < 5000:
        rate = 1.05
    elif balance < 20000:
        # Only runs if balance is also >= 5000
        rate = 1.06
    elif balance < 50000:
        # Only runs if balance is also >= 20000
        rate = 1.07
    return rate
```

Patterns: If-Statements (5)

- Of course, there's always more than one way to solve a problem:

```
def compute_interest(balance):
    ''' Computes interest rate on a bank
    account, based on its balance.  Maximum
    allowed balance is $50,000. '''
    if balance < 5000:
        return 1.05
    if balance < 20000:
        return 1.06
    if balance < 50000:
        return 1.07
```

- This kind of approach is generally frowned upon as bad coding style, and a potential source of bugs

Patterns: Functions Using Loops

- Loops introduce complexities into functions

```
def factorial(n):
    result = 1
    while True:
        result *= n
        n -= 1
    return result
```

- Problem?
  - This function will never terminate
  - The function contains an **infinite loop**

Patterns: Functions Using Loops (2)

- Useful questions when you write a loop:
  - Will this loop actually terminate?
  - Will it terminate at the right time?
    - (And, what is "the right time" for it to terminate?)
  - Does it do the right thing each time through the loop?
    - (And, what is "the right thing" it needs to do?)
  - Are there certain inputs that would make the loop *never* terminate? ☹
- If you don't know these answers, you need to study the problem more.

Patterns: Functions Using Loops (3)

- A better version of factorial?

```
def factorial(n):
    result = 1
    while n != 0:
        result *= n
        n -= 1
    return result
```

- Problem?
  - This function won't terminate for n < 0 ☹
- Generally best to write your code defensively
  - Condition n > 0 ensures that function always terminates

Patterns: For-Loops

- Python `for`-loops are curious beasts
- Can iterate over the actual elements in a list:

```
for elem in lst:
    ... # do stuff with each element
```

- Can iterate over indexes into the list, and extract each element indirectly:

```
for i in range(len(lst)):
    elem = lst[i]
    ... # do stuff with each element
```

Patterns: For-Loops (2)

- Useful questions when writing **`for`**-loops:
  - Can I implement my operation with just the element, or do I need the *index* of each element?
    - e.g. if I am going to remove elements from the list, or if I need to modify the list in-place, I need the index
- Example: pitfall from assignment 2

```
# failed attempt to modify lst in place
for num in lst:
    num *= 2
```

  - **`num`** isn't actually in **`lst`** – it's a separate variable!

Patterns: For-Loops (3)

- Example: pitfall from assignment 2

```
# failed attempt to modify lst in place
for num in lst:
    num *= 2
```

- If I'm going to modify `lst` itself, I need to actually write back into `lst[i]`
  - …this operation requires iterating over the indexes, not just the elements themselves

Patterns: For-Loops (4)

- A better attempt:

```
# second attempt to modify lst in place:
for i in range(len(lst)):
    # Several other ways to write this too,
    # but this version is short.
    lst[i] *= 2
```

- Now the result is being stored back into **`lst[i]`**
- A great example of the critical importance of **understanding the computational model**

Patterns: Recursive Functions

- Recursive functions are fun to write, and often make complicated problems very easy to implement
- Useful questions for recursive functions:
  - What is the base-case for my recursive function?
    - (The case where we can solve the problem immediately)
    - This is where the recursion is *supposed to* terminate
    - Does the recursion actually terminate?!
  - Does the recursive case properly combine its own computation with the results of the recursive invocation?

Patterns: Recursive Functions (2)

- Example: our favorite function

```
def factorial(n):
    return n * factorial(n - 1)
```

- Problems?
  - Properly performs the recursive case…
  - Doesn't include any base-cases! This implementation will never terminate.
- Actually, it will terminate… with an error.

```
RuntimeError: maximum recursion depth exceeded
```

Patterns: Recursive Functions (3)

- Another version:

```
def factorial(n):
    if n > 1:
        return n * factorial(n)
    else:
        return 1
```

- Problems?
  - This time the recursive case doesn't properly reduce the problem (need `n - 1` instead)
  - Again, will recurse forever…
    - At least, for factorials of 2 or more!

Final Words

- Most important thing when programming is to **never give up!**
- Definitely takes time and especially practice to become proficient!
- Everybody has areas that are challenging for them to grasp at first
  - Just keep plugging away at it until you get it
  - And, learn the proper balance of when to ask for help, and when to work through a problem yourself