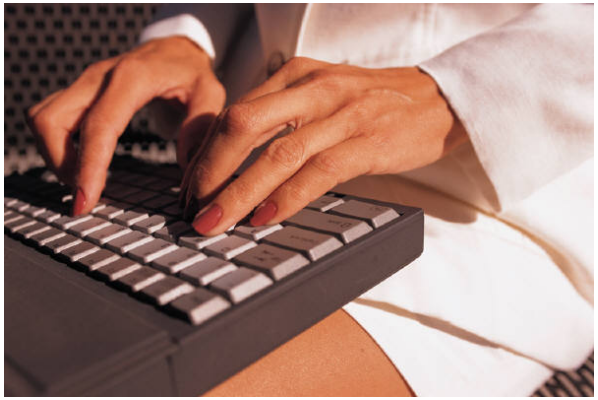
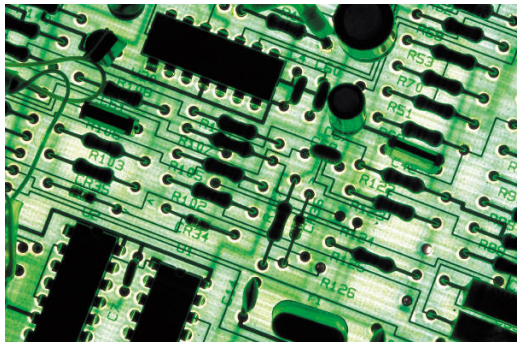
CS 1
Introduction to Computer Programming

Lecture 10: October 23, 2015

How to think about programming





Alternate title (1)

How to design programs

Alternate title (2)

How to design functions

Programming: The Challenge

- Programming can be very hard if you haven't done it before
- There are two main aspects to the difficulty:
  - Learning the language elements (syntax, semantics)
    - What is a function? What is a loop? What is a conditional? How do they work?
    - What are lists? Dictionaries? Tuples? How do they work?
  - Learning how to put the elements together to solve problems
    - How do I design a program that will accomplish a particular task, given just the problem description?

Programming: The Challenge

- In lectures, we've mainly focused on learning the language elements
  - and we'll continue to do so
- In this lecture, we will focus on the *design process*
  - systematic "design recipes"
  - tips, techniques, "rules of thumb"
  - design patterns

Scenario

- You are working on a miniproject for an unnamed intro programming course
  - whose name rhymes with "BS Fun" ☺
- The problem is to write a program which plays the card game blackjack ("21") with the user

Tip: Understand the problem

- You may have never played blackjack before
- The internet is your friend!
- http://en.wikipedia.org/wiki/Blackjack
- Blackjack is a gambling card game
  - Dealer vs. player
  - Goal: To get cards getting as close to 21 but not over
  - Ace counts 1 or 11, Kings/Queens/Jacks count 10, other cards count their face values, suits irrelevant
  - Player plays first, dealer last
  - Dealer must draw below 16 and stand on 17

Tip: Understand the problem

- If you do not understand the problem in detail, you cannot possibly write a solution!
- Need to do research
- Ask TAs if you don't know where to start
- CS 1 assignments should be self-contained for most students
  - read external links if they are supplied!

Tip: Decompose into functions

- Trying to writing an entire program is a big job
- "Where do I start?"
- Basic skill: *decomposing a large problem into smaller problems*
  - applies at multiple levels ("divide and conquer")
- Most obvious example: decompose a program into separate functions
- Much easier to write a function than an entire program!

Tip: Decompose into functions

- How do you decompose a program into functions?
- No hard-and-fast rules for this
  - Programming is a creative activity!
- Rule of thumb: *"Do one thing, and do it well"*
  - Functions should only do one *kind* of thing
  - Don't mix different kinds of tasks into a single function
    - *e.g.* sorting a list and then printing it
    - better to have two functions in that case

Designing functions

- For blackjack game example, what could some of the functions be?

Designing functions

- Blackjack program
  1. create an object representing a deck of cards
  2. shuffle the deck of cards
  3. deal cards to player
  4. compute total count of player's cards
  5. ask user for his play
     - stand (no more cards)
     - hit (one more card)
     - split pairs, double down, etc.
  6. repeat (5) until player busts (> 21) or stands
  7. play for dealer, etc. etc.

Designing functions

- Tip: *Small functions good, big functions bad!*
- It may be possible to write the entire blackjack program as one big function
- It is almost never a good idea to do so
- What is the problem with One Big Function?

Problems with big functions

- Big functions are *hard to understand*
  - many different, often unrelated, things going on
- Big functions are *hard to test*
  - Lots of separate pieces; if something goes wrong, could be a problem in any one of them
- Big functions are *hard to debug*!
  - For the same reason
- Rule of thumb: if your function is >25 lines of code, look for ways to divide it into smaller pieces

Problems with big functions

- Possible objection:
  - "But aren't big functions more efficient than lots of little functions?"
- Answers:
  - No, not really. Maybe a little bit in some cases.
  - At this point, you need to care about *correctness*. Efficiency will come later. We often build a correct, inefficient version of a function or program first and then make it more efficient.
  - Excessive concern about efficiency before correctness is one hallmark of a novice programmer (especially one who wants to be 1337 ("Leet"))

Problems with big functions

- Note: Even if CS 1 assignment says to write a particular function, you are allowed to break that up into more than one function if it's useful to do so
  - more helpful with miniprojects than with exercises, though

Problems with big functions

- Rule of thumb: Ask yourself "What does this function do?"
- If you can't answer that *in a single (short) sentence*, that's a sign that you need to divide the function up into more than one function

Good news!

- For the most part, in CS 1 we tell you exactly what functions we want you to write and what they do
  - so how to divide programs up into functions isn't really a problem in CS 1!
  - (It's a very significant problem in the Real World, though.)
- For the rest of this lecture, we will look at how to write individual functions

Design strategy

- Our approach will involve defining a specific and detailed *design strategy* to use when writing the code for a function
- You will not have to follow every single step of the design strategy for every function you write
- But the harder the function (or the more problems you have with it) the more benefit you will gain from following the steps in the design strategy

Design strategy: outline (1)

- Understand what the function needs to do
- Choose a *name* for the function
- Decide how many *arguments* the function has and their types
  - and choose names for the arguments
- Write a *stub function*
- Decide what is *returned* (if anything) and its type
- Write a detailed *docstring*

Design strategy: outline (2)

- Write one or more *test functions*
- Fill in test cases
- Write *pseudocode* for the function body
  - We'll explain what this is shortly
- Write the code for the function body
- *Test* the function
- *Refactor* (if necessary)
- *Debug* if necessary until all tests pass

Note 1

- The first set of steps should be very easy in CS 1 assignments
  - We've done most of this work for you!
  - Read the problem descriptions carefully!
  - Shouldn't take more than 10 minutes even for hard functions
- Expect to spend most of your time on the second set of steps

Note 2

- Test each function individually!
- Don't write 10 functions that all call each other, then go to TA and say "It doesn't work, help me!"
- TA will answer:
  - "Which functions don’t work?"
  - You had better have an answer!
  - "Uh, I'm not sure..." is not acceptable

Running example

- Our running example will be from a hypothetical blackjack-playing program
- We will write a function that, given as its input a list of cards, will compute the total count of the cards using blackjack rules
- We will follow all the steps in detail

Step: Understand the problem

- The function will receive a list of "cards" and have to compute the count of the cards using blackjack rules
- "Cards" include:
  - numbers: 2…10
  - face cards; Jack, Queen, King
  - Ace
  - Suits (spades, hearts, etc.) are irrelevant in blackjack
- Need to know how cards are *represented* in the program

Data representation

- In many programs, you would have to choose the way cards are represented as Python data
- In CS 1, we generally do that for you
  - Good, because this can be one of the hardest decisions to make!
- We will say that a card is either
  - an integer between 2 and 10
  - one of the strings: `'J'`, `'Q'`, `'K'`, `'A'`
- And a hand of cards is just a list of cards
  - *e.g.* `['A', 'J']`

Step: Understand the problem

- We also need to know how to "add up" cards to get the total count
- In blackjack, suits are irrelevant, so we just add up the card ranks
- Numbers are just added to the sum as themselves
- Jacks, Queens, Kings are all worth 10
- Aces can be either 1 or 11
  - Need to maximize the count without exceeding 21
  - This is the hard part of the whole problem!

Step: Choose a name

- We need to choose a name for our function
- This is much more important than most beginners realize!
- Bad names:
  - `f`
  - `cc` ("count cards"?)
- Good names:
  - `countBlackjackHand` (a bit verbose)
  - `blackjackCount` (nice)
- We'll use `blackjackCount`

Step: Function arguments

- How many arguments does our function need, what are their types, and what name should we use for them?
- This is usually obvious from the problem description in CS 1
- Here, we are adding up a group of cards, represented as a list where every element is either an integer or a one-letter string
- We will call the (only) argument **`cards`**

Step: Stub function

- Now we know enough to write a mock-up of the function, usually called a *stub*
- It has the correct name, the correct number of arguments, and the arguments have the correct names, but that's all
- It doesn't do anything, but it provides a template that we can add to

Step: Stub function

- Here is our stub:

```
def blackjackCount(cards):
    pass
```

- The `pass` statement does nothing; it's a place-holder for where code will eventually go

Step: Stub function

```
def blackjackCount(cards):
    pass
```

- One nice thing about stub functions: we can immediately start to write tests for our function!
  - They will all fail ☺, but then we can add code to make them succeed
  - This is called *"Test-Driven Development"* or *TDD*
- We will do a couple more steps first

Step: Return value(s)

```
def blackjackCount(cards):
    pass
```

- We need to decide what (if anything) this function will return, and what type it will have
- Here, we know that we need to return a positive integer which represents the count of all the cards taken together

Step: Write a docstring

```
def blackjackCount(cards):
    '''
    Compute the sum of a list of cards according
    to the rules of blackjack.
    Arguments:
      cards: a list of cards (2-10 or one of 'JQKA')
    Return value: integer >= 0 representing the
      count of all the cards taken together
    '''
    pass
```

- The docstring summarizes all that we've done so far

Pause

- Some of you are probably getting impatient now
- *"When do we get to the code?"*
- Actually, we've done a lot!
- We have enough information now to write tests for the function
- So... let's write some tests!

Step: Test function

- We'll write a stub ☺ for a test function that will test our **`blackjackCount`** function
- This test function could end up in a test script using the **`nose`** module
- For now, we won't need **`nose`**
- We will call this function...
- ... **`test_blackjackCount`** !

Step: Test function

```
def test_blackjackCount():
    pass
```

- Easy peasy! ☺
- Now comes the real work: writing test cases
- These will constitute an informal *specification* of what we want the function to be able to do
- **If we can't do this, we don't understand the problem!**

Step: Fill in test cases

```
def test_blackjackCount():
    pass
```

- We will use **`assert`** to check that our test cases give the right answer
- Examples:

```
assert blackjackCount([10, 'A']) == 21
assert blackjackCount(['J', 2]) == 12
assert blackjackCount(['A', 'A', 'A']) == 13
```

Step: Fill in test cases

```
def test_blackjackCount():
    assert blackjackCount([10, 'A']) == 21
    assert blackjackCount(['J', 2]) == 12
    assert blackjackCount(['A', 'A', 'A']) == 13
```

- We get rid of the **`pass`** statement because we don't need it anymore (block is not empty)
- Now we could start writing the body of the **`blackjackCount`** function and have a way to test it!

Step: Fill in test cases

```
def test_blackjackCount():
    assert blackjackCount([10, 'A']) == 21
    assert blackjackCount(['J', 2]) == 12
    assert blackjackCount(['A', 'A', 'A']) == 13
```

- However, most programmers who write test cases write far too few of them!
- It's very important to cover the extreme cases
- What would those be here?

Step: Fill in test cases

```
def test_blackjackCount():
    assert blackjackCount([10, 'A']) == 21
    assert blackjackCount(['J', 2]) == 12
    assert blackjackCount(['A', 'A', 'A']) == 13
```

- Extreme cases:
  - no cards!
  - large number of cards (sum > 21)
  - all face cards/no face cards
  - all aces/no aces
  - etc.

Step: Fill in test cases

```
def test_blackjackCount():
    assert blackjackCount([]) == 0
    assert blackjackCount([10, 10, 10]) == 30
    assert blackjackCount(['J', 'Q', 'K']) == 30
    assert blackjackCount([10, 'A']) == 21
    assert blackjackCount(['A', 10]) == 21
    assert blackjackCount(['J', 2]) == 12
    assert blackjackCount(['A', 'A', 'A']) == 13
    assert blackjackCount([2, 3, 4, 5, 6]) == 20
    # etc.
```

Step: Fill in test cases

- If we forget important test cases (or don't write any test cases!) and start writing the body of the function, then our function may not work on all cases it might encounter
- A test function also tells your TA that you understand what the function is supposed to do, even if you can't make it do that yet

Pause

- OK, let's recap where we are:
  - We have a function we want to write
  - We know in general terms what we want it to do
  - We've written a stub
  - We've written a detailed docstring
  - We've written a stub for a test function
  - We've written a number of test cases, which are the specification for the function
- Now we can start writing the code to solve the problem!

Common mistake #1

- The most common beginner's mistake is this:
  - Starting to write the code for the body of the function before doing any of the other steps!
- The code probably won't work
- The students shows the code to a TA
- Is it wrong
  - because the author didn't understand what the function was supposed to do?
  - because there is a bug in the code?
- No easy way to know!

Common mistake #2

- OK, let's assume that all the steps have been completed correctly so far
  - You understand what the function is supposed to do, have written a stub, docstring, and a test function
- Now what do you do?
- You understand *what* the function needs to do, but not *how* to do it
- What's the next step?

Common mistake #2

- Most (> 90%) of students will now immediately start writing the code for the function
- For simple functions, this will often work and is OK
- For hard function, it's a bad idea
- Question: *What makes a function "hard"?*
- Answer: *When it's not obvious how to solve the problem!*

Common mistake #2

- When a student comes to us with a buggy function...
- ... we ask them "How can you solve this problem?"
- We typically get answers like this:
  - *"We need to use a* `for` *loop and a variable* `sum` *to loop over the cards..."*
- What is wrong with answers like this?

Common mistake #2

- *"We need to use a* `for` *loop and a variable* `sum` *to loop over the cards..."*
- It's too specific!
- If you don't understand how to solve the problem without referring to code, you don't understand how to solve the problem!
- But how can you work out a solution without referring to code?
- Answer: Pseudocode!

Technique: Pseudocode

- Just like writing a paper, it helps to outline what a 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)

- An outline of a computer program or function 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.

Technique: Pseudocode (3)

- Where to put your pseudocode: in the bodies of your functions, *before you write them!*
  - Create the function stub, write a docstring, tests, etc. as described previously
  - 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 (4)

- 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

Technique: Pseudocode (5)

- Or, you may be able to write the pseudocode, but...
- 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)

Technique: Pseudocode (6)

- Pseudocode is even useful if it's wrong!
- If you've written pseudocode, you can show it to a TA, and he/she can tell you if it looks good
- If so, go ahead and start translating it into code
- If not, need to revise it

Step: Pseudocode

- Let's write pseudocode for our function

```
def blackjackCount(cards):
    '''<docstring>'''
    # Pseudocode goes here...
    pass
```

Step: Pseudocode

```
def blackjackCount(cards):
    '''<docstring>'''
    # Add up the values of all the cards
    # and return them.
    pass
```

- Problem with this pseudocode?

Step: Pseudocode

```
def blackjackCount(cards):
    '''<docstring>'''
    # Add up the values of all the cards
    # and return them.
    pass
```

- Maybe: *"It's not detailed enough!"*
  - True, but there is something more fundamental.
  - What makes this problem hard?

Step: Pseudocode

```
def blackjackCount(cards):
    '''<docstring>'''
    # Add up the values of all the cards
    # and return them.
    pass
```

- *"It doesn't take aces into account!"*
  - Bingo!
  - An ace can be either 1 or 11.
  - Want to maximize the count of the cards without going over 21. Need to choose values for aces.

Step: Pseudocode (2nd try)

```
def blackjackCount(cards):
    '''<docstring>'''
    # Add up the values of all the cards
    # and return them.  Give aces the largest
    # possible value without going over 21.
    pass
```

- Better, but now you're explaining *what* you will do without explaining *how* you'll do it
- To get further, we'll need to think about aces in more detail (still no code yet!)

Step: Pseudocode

- How should we handle aces?
- If no aces, cards can only have one value, so adding them up is easy
- If one ace, make it 11 if possible, 1 if making it 11 would make total > 21
- If more than one ace:
  - only one can ever be 11 (else total > 21)!
  - add up all aces, using 1 as the value
  - add to count of non-aces
  - add 10 to total if result <= 21, else do nothing

Step: Pseudocode

- We now have an algorithm (systematic procedure) for solving our problem, and we haven't written a single line of Python yet!
- NOTE: Multiple different correct algorithms may exist to solve the problem. Someone else may find a completely different way to solve the same problem. That's OK.
- Let's update our pseudocode based on our algorithm.

Step: Pseudocode (3rd try)

```
def blackjackCount(cards):
    '''<docstring>'''
    # Add up the values of all the non-ace cards.
    # If there are no aces, return that total.
    # Otherwise, add 1 to total for each ace.
    # Then add 10 to total.  If total > 21,
    # subtract 10 from total.  Return total.
    pass
```

- This is good, but we can simplify it a bit.

Step: Pseudocode (4th try)

```
def blackjackCount(cards):
    '''<docstring>'''
    # Add up the values of all the non-ace cards.
    # If there are no aces, return that total.
    # Otherwise, add 1 to total for each ace.
    # If total <= 11, add 10 to total.
    # Return total.
    pass
```

- *Now* we are ready to write some code!

Step: Write the function body

```
def blackjackCount(cards):
    '''<docstring>'''
    # Add up the values of all the non-ace cards.
    # If there are no aces, return that total.
    # Otherwise, add 1 to total for each ace.
    # If total <= 11, add 10 to total.
    # Return total.
    pass
```

- Tip: *Leave the pseudocode in while you write the code until it's done!*

Step: Write the function body

```
def blackjackCount(cards):
    '''<docstring>'''
    # <pseudocode>
    pass
```

- Now we are in the translate-to-Python stage
- First need to "add up the values of all the non-ace cards"
- We are doing something similar to a group of things (cards) – what does that suggest?
- A loop!

Step: Write the function body

```
def blackjackCount(cards):
    '''<docstring>'''
    # <pseudocode>
    total = 0    # need variable to hold total
    for card in cards:
        # Add up value of non-ace cards.
        total += ???
```

- We need to test a card to see if it's an ace or not
- We need to be able to convert a non-ace card to a numerical value
- We also need to keep track of the number of aces

Step: Write the function body

```
def blackjackCount(cards):
    '''<docstring>'''
    # <pseudocode>
    total = 0    # total count of all cards
    aces = 0     # number of aces
    for card in cards:
        if <it's an ace>:
            aces += 1
        else:
            total += <value of card>
```

Step: Write the function body

```
def blackjackCount(cards):
    '''<docstring>'''
    # <pseudocode>
    total = 0    # total count of all cards
    aces = 0     # number of aces
    for card in cards:
        if card == 'A':
            aces += 1
        else:
            total += <value of card>
```

- How do we determine the value of a non-ace card?
- Two cases: integer (2-10) or face card

Step: Write the function body

```
def blackjackCount(cards):
    # ... docstring, pseudocode ...
    total = 0     # total count of all cards
    aces = 0      # number of aces
    for card in cards:
        if card == 'A':
            aces += 1
        elif <card is a face card>:
            total += 10
        else:
            total += card
    # ...rest of code...
```

Step: Write the function body

```
def blackjackCount(cards):
    # ... docstring, pseudocode ...
    total = 0     # total count of all cards
    aces = 0      # number of aces
    for card in cards:
        if card == 'A':
            aces += 1
        elif card in ['J','Q','K']:  # card is a face card
            total += 10
        else:
            total += card
    # Now compute the total.
```

Step: Write the function body

```
def blackjackCount(cards):
    # ... docstring, pseudocode ...
    total = 0     # total count of all cards
    aces = 0      # number of aces
    for card in cards:
        # <body of for loop left out to save space>
    if aces == 0:
        return total
    else:
        ???
```

Step: Write the function body

```
def blackjackCount(cards):
    # ... docstring, pseudocode ...
    total = 0     # total count of all cards
    aces = 0      # number of aces
    for card in cards:
        # <body of for loop left out to save space>
    if aces == 0:
        return total
    else:
        total += aces     # assume each ace == 1
        if total <= 11:
            total += 10   # one ace == 11
        return total
```

Step: Refactor (if necessary)

- *Refactoring* means simplifying/cleaning/DRYing up code without changing its meaning

```
def blackjackCount(cards):
    # ... docstring, pseudocode ...
    total = 0     # total count of all cards
    aces = 0      # number of aces
    for card in cards:
        # <body of for loop left out to save space>
    if aces != 0:
        total += aces     # assume each ace == 1
        if total <= 11:
            total += 10   # one ace == 11
    return total
```

Final version

```
def blackjackCount(cards):
    # ... docstring ...
    # pseudocode no longer necessary, so remove it!
    total = 0     # total count of all cards
    aces = 0      # number of aces
    for card in cards:
       if card == 'A':
          aces += 1
       elif card in ['J','Q','K']:  # card is a face card
          total += 10
       else:
          total += card
    # ...<continued on next slide>...
```

Final version

```
def blackjackCount(cards):
    # ... <previous slide> ...
    if aces != 0:
        total += aces     # assume each ace == 1
        if total <= 11:
            total += 10   # one ace == 11
    return total
```

Finishing up

- We still have to
  - test our function
  - debug if necessary until all tests pass
- Testing is easy!
  - We already wrote our test function, right?
- Debugging will be the subject of a future lecture

Summary

- Programming is hard!
- Programming is *much* harder if you just randomly try things and hope that something works!
  - The "monkey on a typewriter" methodology
- We've given you a systematic design methodology for writing functions
- You will need to adapt this to your own needs and to particular functions
- Now you know "where to start"!

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
- Learn the proper balance of
  - when to ask for help
  - when to work through a problem yourself!