Thinking Like A Programmer

Objectives

Welcome to the workshop! I'm very excited that you've chosen to learn more about computers and programming with me, and it's my sincere hope that you leave knowing more than you came in with, fully prepared to start learning more on your own about modern programming languages.

My primary objective is to give everyone in the workshop a set of mental "tools" for solving problems elegantly and efficiently. By the time we're done, you should be able to:

- Approach problems in a methodical and logical way
- Break large problems into smaller problems that can be solved in steps
- Apply the programming language concepts learned to any other programming language
- Learn how to perform research on the Internet to find further tutorials and references

Materials

In order to hit the ground running, we recommend using the following;

- A MacBook of any hardware specification, running Mac OS 10.4 or greater

Apple computers installed with the Mac OS X operating system already have every software tool needed to start learning how to program right away!

If you have a PC, the self-installing Python version you need to download can be found here:

http://www.python.org/ftp/python/2.7.2/python-2.7.2.msi

Next you will need to follow the instructions under the sections "Finding python.exe" and "Finding your script files", as written in the tutorial found here:

http://www.imladris.com/Scripts/PythonForWindows.html

Syllabus

1. Welcome to Workshop
    1.1. About NWC
    1.2. About Me
    1.3. Review of Course Outline

2. Introduction to Computers - Presentation
    2.1. What Do You Know?
    2.2. The History of Computers - pre-1940
    2.3. Alan Turing & WWII
    2.4. Digital Computers
        2.4.1. Binary Numbers
        2.4.2. What is a Digital Computer?
    2.5. What is UNIX?
        2.5.1. Operating Systems

3. Under Your Computer's Hood
    3.1. Terminal
        3.1.1. Command Line Interface Demo
    3.2. Unix Commands
    3.3. Editing Files and Your Shell's Settings
        3.3.1. Using Emacs
        3.3.2. Alias rm

4. Programming Python (at last!)
    4.1. A Quick Note - 2.7 Versus 3.0
    4.2. The Basics of Python
        4.2.1. The Interpreter
        4.2.2. Numbers and Truth Values, Operators and Expressions
        4.2.3. Types, Characters and Strings
        4.2.4. Types and Variables

Slide Notes

Each paragraph break below represents a single slide in the presentation. These slide notes are for the Prezi presentation located online at:
http://prezi.com/oscegmctm16i/a-conceptual-history-of-computing/

I'm here to tell you you don't need to be a wizard to learn how to program.

You don't need to go to some school for witchcraft and wizardry for years and years...

Where you're going to learn all kinds of strange incantations...

And you end up working for Google making balls levitate.

So the first step: Tell me what you already know about computers so we can clear up misconceptions and see what our baseline is.

What a lot of people don't know is that computers have existed for thousands and thousands of years.

People made calculations with other kinds of machines, what we call "analog computers"

The Abacus - this device was developed nearly 5,000 years ago for rapidly adding numbers together for all kinds of purposes: accounting, inventory, anything that we need to calculate today. It spread all over the ancient world, across the cradle of civilization and Asia. Interestingly, today's computers function in a similar way - in fact even up until just a few decades ago, a human computer with an abacus could beat a modern computer.

The Antikythera mechanism - this was discovered at the bottom of the Mediterranean last century. It was later realized this was an ancient Grecian device over 2000 years old, which was designed to predict astronomical events and ceremonial dates such as when the olympics should be held.

As recently as two centuries ago, we had Charles Babbage of England and his calculating engines. Unfortunately, he never received the funding to produce his masterpiece - this is what he would have built if he had the resources, it was commissioned by Nathan Myhrvold, a former Microsoft CTO who has made a fortune exploiting software patents.

The future was written during the Second World War, when the Germans created a calculating machine codenamed "Enigma" for encoding their secret communications. British mathematician Alan Turing cracked the Enigma, and in the process began his work on the fundamental theories of computer science.

Contemporaneously with Alonzo Church he devised the theory that anything a mechanical calculator could do, could also be done using a new kind of device called an electro-mechanical or digital computer.

Today's modern computers are all digital computers, meaning they use electricity to compute; just think of the difference between an analog watch and a digital watch.

Fundamentally, all a digital computer does is add numbers together using electricity.

A number is represented by turning a circuit on or off - which translates to 1 or 0 respectively. A number system that has two digits is called "binary"; we're used to using 10 digits in our number system, so we call them "decimal" numbers. A number system with 16 digits is "hexadecimal".

Binary digits, or "bits" are added together by a computer; a collection of 8 bits is a byte. Binary numbers can be represented by collections of circuits, which is all a computer really is, and any binary number can be translated into any number.

The important thing to take away from this is that ANYTHING can be a number. File, photo, music - just a big number.

But how does a modern digital computer work? The best way I can think of to describe it is that it's like a kitchen.

First, note that CPUs are measured in "Hertz" - instructions per second. Giga means "billion". The CPU is like a chef - it does stuff.

The Hard Drive is like the fridge - it's the place for long term storage that you fetch ingredients from.

The Memory is like the counter - the working area that is "closest" to the chef where stuff is done. The bigger the counter, the more working area you have and the more you can done at once.

Hard drives and memory are measured in "bytes" so we can measure the size of the numbers they store.

And now the genius who created computers as we know them today!

No, not that guy. He wasn't really that inventive, he just knew how to hire smart people & license technology, like Thomas Edison.

Douglas Engelbart @ Stanford - most modern interface elements invented in the 1960s

XEROX PARC - improved the GUI

IBM - disc drives, databases, DRAM

AT&T Bell Labs - many technology advances, created languages, communications theory

Still one problem: computers could only do one thing at a time. one terminal -> one program

Bell Labs worked on MULTICS so multiple users could perform tasks on a single computer system, it failed, and then the underground project Unix was born

Dennis Ritchie, inventor of C language and Unix co-creator - deserves our remembrance and respect

Unix led to creation of MS-DOS (later MS Windows) and then Mac OS X. Linux is derived from Unix and runs on more devices and computers than anything else.

Operating system takes care of all tasks for multiple hardware and software entities on a system

Hardware at the lowest level can talk to each other but nothing coordinates them

System level, or kernel, has drivers so devices can communicate, other utilities for performing tasks on devices

Applications level is where user programs run, they are organized into jobs by the kernel, they time-share the processor and other hardware. The user can communicate directly with the operating system by running a shell, and that is where we will begin today.

Standard UNIX Commands

Nearly every system that is based on Unix (Linux, Mac OS X, HPux, AIX, Solaris, to name a couple) has these basic commands available to users in the shell of their choice. If you can get access to a terminal or command line prompt (usually signified by a `$` symbol), you can probably enter any of these commands at the shell's prompt and expect the same results. You'll need to know all of these at some point to properly navigate a Unix based system. If you ever need help for a command, just run the "man" command with a command name, as described below.

In many cases, when multiple options for a command are available, you can combine them together, as you would below to list all files in long format.

```
$ ls -la
```

The very first thing a new shell user should do is edit their configuration file and create an alias for the rm command. On Mac OS X, the default shell is the "bash" or "Bourne Again" shell - you should edit or create a file called .bash_profile and add the line:

```
alias rm='rm -i'
```

This as explained below will ask you for confirmation every time you try to delete a file. To activate this alias, simply type in the command:

```
$ source .bash_profile
```

One last tip: using the "up" cursor arrow will display previous commands that you've used since you started the shell. Very useful!

Command: ls

Options: -l for long format, -a for all files

Description: This command lists all the files in your present working directory. Can optionally specify a partial filename with wildcard as in: `*.py` or `test*` to list files with certain names.

Command: more (path/filename)

Options: none usually needed

Description: Displays the contents of a file to the screen.

Command: mv (path/filename) (path/filename)

Options: none usually needed

Description: This will move a file from one directory to another.

Command: cp (path/filename) (path/filename)

Options: none usually needed

Description: This command lists all the files in your present working directory.

Command: rm (path/filename)

Options: -i to prompt yes or no to confirm

Description: This command lists all the files in your present working directory. It's best practice to alias rm to rm -i in your shell configuration files to always prompt when deleting.

Command(s): gzip, gunzip (path/filename)

Options: none usually needed

Description: One particular pair of commands which will compress and decompress files respectively. Files will end in .gz extension when compressed.

Command: tar (path/filename)

Options: usually specified with -xvf for extracting

Description: Another compression/decompression utility, often used when distributing software packages for installation. Most times you will merely be running `tar -xvf (path/filename)` to extract a series of files and directories.

Command: mkdir (path/ directoryname)

Options: none usually needed

Description: This allows you to create a new, empty directory in your current directory.

Command: rmdir (path/ directoryname)

Options: none usually needed

Description: Removes a directory.

Command: cd (path/directoryname)

Options: none usually needed

Description: This command changes your current directory to the specified path and directory. If you want to go "up" one directory level, you can just type: `cd ..`

Command: pwd

Options: none usually needed

Description: This command tells you current directory path - "present working directory".

Command: Ctrl-z

Options: none

Description: This keystroke command will "suspend" whatever job you are currently running in the shell and return you to a command prompt.

Command: fg

Options: none

Description: When there is a suspended job, typing this at the command prompt will return it to the "foreground".

Command: man (commandname)

Options: none usually needed

Description: Displays the exhaustively comprehensive help file for the command, listing usage, options, and verbose descriptions of what the command can do.

Standard Emacs Commands

Emacs is a very powerful text editor included in most Unix systems (it actually does much more than just edit text files, but that's mainly what people use it for). It is a bit of a holdover from the early days when programmers used "dumb terminals" and had to use certain keystroke combinations to produce particular effects. Today people rarely use the "control" and "function" keys on a keyboard - unless they're power users, or programmers! Emacs is easy to use and extremely powerful when you know what just a few of the commands are.

To start emacs, just type:

```
$ emacs filename.py
```

And you can edit multiple files at the same time in several "buffers" on the screen by just entering multiple filenames. As with most text editors, you can navigate your cursor around the page using the arrow keys. Type in text, and it appears on the screen. Interesting information appears at the bottom of the screen, such as the filename, the percentage of the file you are currently looking at, and the line number. After closing a file, you'll usually see a copy of it with a ~ symbol at the end; this is an automatic backup of the previous file just in case you want to go back to it.

You'll notice that because the file has a .py extension, the bottom of the screen will say "(Python)" and that you'll see different colors for different words in the file - this is because emacs can detect the file type and display appropriate color cues for the type of programming language you're working with. It will also do other helpful things like indenting lines for you when you hit the tab key, and showing you where parentheses match and mismatch - just check the bottom of the terminal to see if any messages pop up. You can even edit multiple files at once on the same screen and switch your cursor back and forth between them!

Below is a list of the most useful and popular commands you'll need to know. When you see a command with "Ctrl-" at the beginning of it, it means you should hold down the "control" key and then hold down another key. This is exactly the same thing one does when performing a "copy, paste" operation - on PC, that's Ctrl-c, Ctrl-v; on an Apple computer that's command-c, command-v.

When you see the "M-x" or "Meta-x" type of command being referenced, that means you should hold down "control", then left bracket "[" and then let them go and type "x". Or, in other words: Ctrl-[, x

Command: Ctrl-x, Ctrl-c

Description: These two key commands one after the other will close your current emacs session. You'll be prompted for whether you want to save changes to the file, so enter "y" or "n" to do so.

Command: Ctrl-x, Ctrl-s

Description: Saves the current file while keeping the emacs session open. Will prompt for yes or no.

Command: Ctrl-k

Description: "Kill" the current line. Position your cursor to where you want to "cut" the current line and hit Ctrl-k. Keep hitting Ctrl-k or hold it down to kill multiple lines; lines for any particular "kill" are saved temporarily for you - but take note! If you kill lines, then move your cursor and kill lines again, you lose all the lines you cut out with the previous kill.

Command: Ctrl-y

Description: "Yank" back any lines that you've just killed. The equivalent of "paste" but only returns the lines that have been most recently killed. Be careful!

Command: Ctrl-x, o

Description: Switches to another visible "buffer". Useful when editing multiple files on screen at the same time; Kill and yank lines in one file, then switch to the other file and yank them out to copy and paste a bunch of code.

Command: Ctrl-x, i

Description: Insert the contents of a file into the buffer / file you're currently editing. You'll be prompted to type in the path and name of the file at the bottom of the screen - hitting the tab key will autocomplete file names just the same way as in the shell.

Command: M-x replace-string

Description: Replaces every single instance of one string in your file with another specified string. In other words, you can change the text "_value" to "_points" everywhere in your code. Useful, but again, be careful you don't replace things you don't want to replace!

Command: M-x goto-line [number]

Description: Enter the number of the line of code you want to go to, and emacs takes you there.

Command: Ctrl-[, < and Ctrl-[, >

Description: Move the cursor to the beginning of the file with the < sign, and move the cursor to the end of the file with the > sign.

Further Reading

Learning Python
Lutz, Mark. O'Reilly Media, Inc., October 2009

An excellent guide and comprehensive learning reference. O'Reilly is a top notch publisher, you generally cannot go wrong with any O'Reilly book.

The C Programming Language
Kernighan, Brian and Ritchie, Dennis. O'Reilly Media, Inc., 1978 (many later editions)

A.K.A. "K&R" for Kernighan and Ritchie. Nearly every device in the world that isn't a fully general purpose computer is running an embedded firmware / real time operating system that is built with C. All Unix and Unix-derived operating systems are written in C. C++ and Microsoft's C# (putatively) derive from C. Apple's iOS for iPhone & iPad is written in Objective-C, which is derived from C. If you really want to become a true master of all things programming, you need to learn C.

C Problem Solving And Programming
Barclay, Kenneth. Prentice Hall, 1990 (has later editions)

Another terrific text for learning C. My personal favorite.

Structure And Interpretation Of Computer Programs
Abelson, Harold and Sussman, Gerald and Julie. MIT Press, 1985 (has later editions)

The primary text for computer science students at MIT. It teaches programming from a somewhat different perspective than most, as it teaches functional programming languages versus procedural. Nearly all programming languages, Python and C included, are procedural (or imperative), while LISP, Haskell and Scheme are mostly functional. This is truly for advanced programmers, as it really does represent a wholly different way of "thinking like a programmer".

Godel, Escher, Bach: An Eternal Golden Braid
Hofstadter, Douglas. Basic Books, 1979 (many later editions)

A mind-bending journey into the strangeness of the universe, a lyrical exploration of the conundrums of the fundamental theories of computer science mixed with music and mathematics. You really need to try reading it. This is basically the ultimate zen holy grail of programming.

Links

http://www.python.org/
The official Python web site - everything you need to get everything Python.

http://docs.python.org/
The official documentation for the Python language.

http://www.diveintopython.net/
A great GNU licensed tutorial text that can take you further with the Python language.

http://stackoverflow.com/
Stack Overflow is an invaluable resource for getting feedback and advice from other programmers. Google searches will usually return results from this site if you search for specific enough keywords.

Vocabulary

In this section we define a number of terms related to programming.

shell
Definition: The user environment where operating system commands can be executed at a command line prompt.

variable
Definition: A named "storage vessel" in a program which can store a value and changes every time a new value is assigned to it.

assignment
Definition: The act of storing a value into a variable, using the assignment operator `=`

operator
Definition: Usually a mathematical symbol, such as: `+, -, /, *` where the slash is division and the asterisk is multiplication. Operators can also be used in multiple ways depending on the types being operated upon

type
Definition: The nature of the variable or value in a programming language. Types in Python include integer, string, character, list, and so on. Python is considered a "loosely typed" language because variables and values can change their type without causing errors in execution.

execute
Definition: To start a program.

Code Examples

(guess_comments.py)

```
# Choose a random number

# Keep prompting for input until the game is over

    # Check to see if input is a number

        # Convert number to an integer

        # Check to see if it is an integer between 1 and 100 inclusive

        # If it is a good number guess, but not the right number, give hints

            # Give lower hint

            # Give higher hint

            # The number equals the guess!

    # Return an error
```

(guess.py)

```
import random

# Choose a random number
the_number = random.randint(1, 100)

# Keep prompting for input until the game is over
while True:
    number_guess = raw_input("Guess an integer between 1 and 100: ")

    # Check to see if input is a number
    if (number_guess.isdigit()):
        # Convert number to an integer
        number_guess = int(number_guess)

        # Check to see if it is an integer between 1 and 100 inclusive
        if (number_guess < 1 or number_guess > 100):
            print "Bad guess! Must be a number between 1 and 100."
            continue
        else:
            # If it is a good number guess, but not the right number, give hints
            if (the_number < number_guess):
                # Give lower hint
                print "Guess lower!"
            elif (number_guess < the_number):
                # Give higher hint
                print "Guess higher!"
            else:
                # The number equals the guess!
                print "You got it!!"
                break
    else:
        # Return an error
        print "Bad guess! Must be an INTEGER NUMBER between 1 and 100."
        continue
```

(craps_comments.py)

```
# Get the bet from the player - pass or don't pass

# Keep prompting for rolls until the game is over

    # Let the player hit return to roll the dice

    # Roll two dice

    # If it is the first roll, and it is not 7 or 11, declare the point

            # If it is 7 or 11 on the first roll and player bet pass, they win

            # If it is 7 or 11 and player bet don't pass, they lose

        # Set the point

        # Check the result to see if it matches the point

                # If it is 7 or 11 and player bet pass, they lose

                # If it is 7 or 11 and player bet don't pass, they win

        # The result matches the point

                # The player makes the point and bet pass - a win

                # The player makes the point and bet don't pass - they lose

        # No result matches point or is 7, 11 yet - keep rolling!
```

(craps.py)

```
import random

# Get the bet from the player - pass or don't pass
while True:
    player_bet = raw_input("Would you like to bet pass or don't pass? (enter p or dp): ")

    if ((player_bet == "p") or (player_bet == "dp")):
        break

player_point = 0

# Keep prompting for rolls until the game is over
while True:
    # Let the player hit return to roll the dice
    discard_input = raw_input("Press Return to roll...")

    # Roll two dice
    die_1 = random.randint(1, 6)
    die_2 = random.randint(1, 6)

    result = die_1 + die_2

    # If it is the first roll, and it is not 7 or 11, declare the point
    if (player_point == 0):
        if ((result == 7) or (result == 11)):
            if (player_bet == "p"):
                # If it is 7 or 11 on the first roll and player bet pass, they win
                print "You rolled %d - you win!" % (result, )
            elif (player_bet == "dp"):
                # If it is 7 or 11 and player bet don't pass, they lose
                print "Ouch, you rolled %d and bet \"don't pass\" - you lose. Game Over." % (result, )
        else:
            # Set the point
            player_point = result

            print "You rolled %d, %d - your point is %d" % (die_1, die_2, player_point)
    else:
        print "You rolled %d, %d - result of %d, your point is %d" % (die_1, die_2, result, player_point)

        # Check the result to see if it matches the point
        if ((result == 7) or (result == 11)):
            if (player_bet == "p"):
                # If it is 7 or 11 and player bet pass, they lose
                print "Ouch, you rolled %d and bet \"pass\" - you lose. Game Over." % (result, )
```

```
        elif (player_bet == "dp"):
            # If it is 7 or 11 and player bet don't pass, they win
            print "You rolled %d - you win!" % (result, )

        break
    elif (result == player_point):
        # The result matches the point
        if (player_bet == "p"):
            # The player makes the point and bet pass - a win
            print "You rolled %d - you win!" % (result, )
        elif (player_bet == "dp"):
            # The player makes the point and bet don't pass - they lose
            print "Ouch, you rolled %d and bet \"don't pass\" - you lose. Game Over." % (result, )

        break
    else:
        # No result matches point or is 7, 11 yet - keep rolling!
        print "Keep rolling!"
```

(blackjack_comments.py)

```
# Define what a card is; each card has a name, a point value, and a suit

    # Return the card's name and suit for printing

    # Return the card's name

    # Return the card's point value


# Create a deck of cards

    # Use a double ended queue structured list for the deck

    # For each suit, create a card with each of the name and point entries


# Select the top card from the deck


# Calculate the points for a hand

    # Check to see if hand got dealt an Ace and whether 11 points or 1 point

    # For each card, add together all the points

        # Check for Aces, get the name of the card

    # How to determine if Aces are worth 1 or 11
    # A - 1 or 11
    # AA - 2 or 12
    # AAA - 3 or 13
    # AAAA - 4 or 14

    # Add 10 points to the total if it doesn't bust the hand


# First create the deck

# Shuffle the deck

# Print a welcome message

# Deal two cards to the player
```

```
# Deal two cards to the dealer

# Play the game until the player or dealer finish

    # Check to see if input is hit or stand

        # If hit, deal a card

            # Check to see if player busts

            # Check to see if player wins

        # If stand, player is done, deal dealer's cards

            # Dealer must hit if points are <= 16 and must stand on > 16

                # Dealer must hit

                    # Check to see if dealer busts

                    # If dealer gets 21, house wins

                # Dealer has to stand, check to see who wins

                    # Player wins with more points

                    # Or there is a tie

                    # Or the dealer wins with more points

    # Or maybe there is bad input
```

(game_library/card.py)

```
from collections import deque


# Define what a card is; each card has a name, a point value, and a suit
class Card:
    def __init__(self, this_card_name, this_card_points, this_card_suit):
        self.card_name = this_card_name
        self.card_points = this_card_points
        self.card_suit = this_card_suit

    # Return the card's name and suit for printing
    def get_card(self):
        return unicode(self.card_name) + self.card_suit

    # Return the card's name
    def get_name(self):
        return unicode(self.card_name)

    # Return the card's point value
    def get_points(self):
        return self.card_points

# Create a deck of cards
def create_deck():
    suit_list = [unichr(0x2665), unichr(0x2666), unichr(0x2663), unichr(0x2660)]
    name_points_dict = {"A":1, "2":2, "3":3, "4":4, "5":5, "6":6, "7":7, "8":8, "9":9, "10":10, "J":10, "Q":10,
"K":10}

    # Use a double ended queue structured list for the deck
    deck_list = deque([])

    # For each suit, create a card with each of the name and point entries
    for each_suit in suit_list:
        for each_entry in name_points_dict.keys():
            new_card = Card(each_entry, name_points_dict[each_entry], each_suit)
            deck_list.append(new_card)

    return deck_list

# Select the top card from the deck
def deal(this_deck):
    dealt_card = this_deck.popleft()

    return dealt_card
```

```
# Calculate the points for a hand
def calculate_points(this_hand):
    # Check to see if hand got dealt an Ace and whether 11 points or 1 point
    total_points = 0
    int_ace_count = 0

    # For each card, add together all the points
    for each_card in this_hand:
        total_points += each_card.get_points()

        # Check for Aces, get the name of the card
        this_card_name = each_card.get_name()

        if (this_card_name == "A"):
            int_ace_count += 1

    # How to determine if Aces are worth 1 or 11
    # A - 1 or 11
    # AA - 2 or 12
    # AAA - 3 or 13
    # AAAA - 4 or 14

    if (int_ace_count > 0):
        # Add 10 points to the total if it doesn't bust the hand
        if ((total_points + 10) <= 21):
            total_points += 10

    return total_points
```

(blackjack.py)

```
import random

from game_library.card import *



# First create the deck
deck = create_deck()

# Shuffle the deck
random.shuffle(deck)

dealer_hand = []
player_hand = []

dealer_total = 0
player_total = 0

bool_player_done = False

# Print a welcome message
print u"Welcome to blackjack! Highest to 21 wins. Dealing for you..."

# Deal two cards to the player
player_card = deal(deck)
player_hand.append(player_card)
player_total = calculate_points(player_hand)

print u"You have: " + player_card.get_card()

player_card = deal(deck)
player_hand.append(player_card)
player_total = calculate_points(player_hand)

print u"You have: " + player_card.get_card() + u", total of " + unicode(player_total)

# Deal two cards to the dealer
print u"Now dealing for the dealer..."

dealer_card = deal(deck)
dealer_hand.append(dealer_card)
dealer_total = calculate_points(dealer_hand)

print u"Dealer deals one card up, dealer has: " + dealer_card.get_card() + u", total of " +
unicode(dealer_total)
```

```
dealer_card = deal(deck)
dealer_hand.append(dealer_card)
dealer_total = calculate_points(dealer_hand)

print u"Dealer deals another card down."

# Play the game until the player or dealer finish
while True:
    if (not bool_player_done):
        player_action = raw_input("Do you want to hit or stand? (h or s): ")
    else:
        player_action = "s"

    # Check to see if input is hit or stand
    if ((player_action == "h") and (not bool_player_done)):
        # If hit, deal a card
        player_card = deal(deck)
        player_hand.append(player_card)
        player_total = calculate_points(player_hand)

        print u"Player dealt: " + player_card.get_card() + u", total of " + unicode(player_total)

        if (player_total > 21):
            # Check to see if player busts
            print "Busted! Game over."
            break
        elif (player_total == 21):
            # Check to see if player wins
            print "Player has 21!"
            bool_player_done = True
    elif (player_action == "s"):
        # If stand, player is done, deal dealer's cards
        string_dealer = u"Dealer reveals his cards: "

        for each_card in dealer_hand:
            string_dealer += each_card.get_card() + u", "

        dealer_total = calculate_points(dealer_hand)
        string_dealer += u"total of " + unicode(dealer_total)

        print string_dealer

        while True:
            # Dealer must hit if points are <= 16 must stand on > 16
            if (dealer_total <= 16):
                # Dealer must hit
```

```
            print "Dealer hits..."

            dealer_card = deal(deck)
            dealer_hand.append(dealer_card)
            dealer_total = calculate_points(dealer_hand)

            print u"Dealer has: " + dealer_card.get_card() + u", total of " + unicode(dealer_total)

            if (dealer_total > 21):
                # Check to see if dealer busts
                print "Dealer busted! Player wins!!"
                break
            elif (dealer_total == 21):
                # If dealer gets 21, house wins
                print "Dealer wins! Game over."
                break
        else:
            # Dealer has to stand, check to see who wins
            print "Dealer stands."

            if (player_total > dealer_total):
                # Player wins with more points
                print "Player wins with %s!!" % (str(player_total), )
            elif (player_total == dealer_total):
                # Or there is a tie
                print "Push! Looks like a tie - Game over."
            else:
                # Or the dealer wins with more points
                print "Dealer wins with %s! Game over." % (str(dealer_total), )

            break

    break
else:
    # Or maybe there is bad input
    print "Bad input! Must be h or s to hit or stand respectively."
    continue
```

Credit & License