Coding For Beginners

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 with, fully prepared to learn more on your own about programming languages.

My primary objective is to give everyone in the workshop a set of mental "tools" and computer programming knowlege that will help with solving problems elegantly and efficiently. By the time we're done today, you should be able to:

(1) Understand and explain the basics of how computers work
(2) Use operating system user commands via the command line interface
(3) Install and configure programming tools and third party software libraries
(4) Learn how to perform research on the Internet to find further tutorials and references
(5) Consider all the variables of a problem in a methodical and logical way
(6) Break large problems into smaller problems that can be solved in incremental steps
(7) Apply the fundamental programming concepts learned to any other programming language

This may sound like a lot, but don't worry — we will cover it all! Feel free to interrupt and ask questions and for elaboration on anything and everything. There are no embarrassing questions, and there is no judgement! Everyone starts at the same place with computers, knowing nothing about them or about programming. Nobody is "born with it" and everyone can learn, it just takes determination, patience, and practice.

Think of learning how to read and write software as the same as learning how to read and write your native language. People start by learning how to speak, then they learn letters, then words, then sentences, and eventually they can write or read anything and become literate. That is exactly what you are doing today, you are taking the first steps towards becoming literate in another language - a language that will help you process information and solve problems!

Materials

In order to hit the ground running, I recommend fulfilling the following computer requirements:

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

Apple computers running the Mac OS X operating system already have every software tool needed to start learning how to program right away! The Homebrew package manager is only used with Mac OS X. The code editor we will be using, Sublime Text, is available in both Mac OS X and Windows.

PLEASE NOTE: If you choose to use a Windows computer, you will not be able to follow along with any of the open source UNIX command or package manager instructions.

If you have a Windows PC, the self-installing Python version you need to download can be found here: https://www.python.org/ftp/python/2.7.8/python-2.7.8.msi

Next you will need to follow the instructions appropriate for your Windows OS version to set your Windows PATH:

Windows 7/8

- Select Computer from the Start menu
- Choose System Properties from the context menu
- Click Advanced system settings > Advanced tab
- Click on Environment Variables, under System Variables, find PATH, and click on it.
- In the Edit windows, modify PATH by adding:

  `C:\Python27\;C:\Python27\Scripts;`

  to the value for PATH. If you do not have the item PATH, you may add a new variable with **`PATH`** as the name and **`C:\Python27\;C:\Python27\Scripts;`** as the value.
- Reopen Command prompt window, and type in **`python`** to check if the PATH variable is set correctly.

Windows XP

- Start -> Control Panel -> System -> Advanced
- Click on Environment Variables, under System Variables, find PATH, and click on it.
- In the Edit windows, modify PATH by adding:

  `C:\Python27\;C:\Python27\Scripts;`

  to the value for PATH. If you do not have the item PATH, you may add a new variable with **`PATH`** as the name and **`C:\Python27\;C:\Python27\Scripts;`** as the value.
- Reopen Command prompt window, and type in **`python`** to check if the PATH variable is set correctly.

Windows Vista

- Right click My Computer icon
- Choose Properties from the context menu
- Click Advanced tab (Advanced system settings link in Vista)
- In the Edit windows, modify PATH by adding:

  `C:\Python27\;C:\Python27\Scripts;`

  to the value for PATH. If you do not have the item PATH, you may add a new variable with **`PATH`** as the name and **`C:\Python27\;C:\Python27\Scripts;`** as the value.
- Reopen Command prompt window, and type in **`python`** to check if the PATH variable is set correctly.

Syllabus

1. Welcome to Workshop
   - 1.1. About The Classroom Space
   - 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. Modern Digital Computer Architecture
   - 2.6. The Computer Renaissance
     - 2.6.1. Modern Computer Origins
     - 2.6.2. What is UNIX?
   - 2.7. Operating Systems

3. Under Your Computer's Hood
   - 3.1. Terminal
     - 3.1.1. Command Line Interface vs. Graphical Interface Demo
   - 3.2. Unix Commands
   - 3.3. Editing Files and Your Shell's Settings
     - 3.3.1. Installing and Using Sublime Text
     - 3.3.2. Displaying and Editing Hidden Files (Mac OS X)
     - 3.3.3. Your bash profile, and "alias rm" (Mac OS X)

- **4.** Expanding Your Programming Toolbox
  - **4.1.** Using a Package Manager
    - **4.1.1.** Installing Homebrew
      - **4.1.1.1.** Xcode command line tools - just install & continue
      - **4.1.1.2.** Why "brew install python"?

- **5.** LUNCH BREAK

- **6.** Programming Python (at last!)
  - **6.1.** The Interpreter - Statements and Expressions, Evaluation
  - **6.2.** Numbers and Types
    - **6.2.1.** Operators ( ), **, *, /, %, +, - [PEMDAS]
    - **6.2.2.** Integers, Long integers, Floats (Floating Point Numbers)
    - **6.2.3.** Strings and Characters
  - **6.3.** Variables and Assignment of Values
  - **6.4.** Truth and Logic
    - **6.4.1.** True, False
    - **6.4.2.** and, or, not
    - **6.4.3.** !=, ==
    - **6.4.4.** <, >, >=, <=
  - **6.5.** Control Flow
    - **6.5.1.** Conditional - if, elif, else
    - **6.5.2.** Loop - while, for
  - **6.6.** Collections
    - **6.6.1.** list, dictionary, tuple, set
  - **6.7.** Functions and Arguments
  - **6.8.** A Few More Things Before We Start
    - **6.8.1.** "Blocks" of Code - Indentation Defines Blocks!
    - **6.8.2.** python filename.py - Executes Your Code

Slide Notes

Each paragraph of text below represents a single slide in the presentation. These slide notes are for the Prezi presentation located online at:
http://prezi.com/osccgmctm16i/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 digital 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.

The Difference Engine - 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, Turing 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". The ancient Mayans used a "vigesimal" number system that had 20 digits, and the Babylonians used a "sexagesimal" system with 60 digits!

Binary digits, or "bits" are added together by a computer; a collection of 8 bits is 1 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 do the components of 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. Hard drives and memory are measured in "bytes" so we can measure the size of the numbers they store.

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.

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

Still one problem: computers could only do one thing at a time. 1 terminal -> 1 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. GNU/Linux is derived from Unix and runs on more devices and computers than any other operating sytem.

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
```

A couple of other tips: using the "up" cursor arrow will display previous commands that you've entered since you started the shell. You can also use the Tab key to "autocomplete" commands or filenames - get in the habit of always hitting the Tab key. Very useful!

Command: `ls`

Options: -l for long format, -a for all files, many more options

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: `cd (path/directoryname)`

Options: none usually needed

Description: This command changes your current directory to the specified path and directory. In UNIX, the topmost directory in the file system is the "root" directory and is referred to as a

single forward slash, like so: **`/`** . Your user home directory is referred to as "tilde" or **`~`** . Typing just **`cd`** will always return you to your home directory. The directory you are currently in can be referred to as a single dot: **`.`** and you would use this to run a script or executable, like so: **`./script.sh`** . 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: `more (path/filename)`

Options: none usually needed

Description: Displays the contents of a file to the screen. Use 'b' keystroke to move back, spacebar to move forward, cursor keys to move line by line, 'q' to quit.

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: `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(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: `Ctrl-c`

Options: none

Description: This keystroke command will "interrupt" whatever job you are currently running in the shell and return you to a command prompt. The execution will stop completely.

Command: `Ctrl-z`

Options: none

Description: Sometimes a job will run for a long time, and you just want to pause it to return to a command prompt; 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. It works with all the same keystrokes that the more command does! You can learn more about how man works by entering **`man man`**

Setting Your Computer Up For Programming

Most computers and laptops are sold with an operating system already installed. In the case of Apple computers, that operating system is Mac OS X, which happens to be a UNIX derived system also called Darwin. Apple's version of UNIX is "slimmed down" and doesn't contain every piece of software that comes with a more standard version of UNIX. Windows PCs and laptops run the Microsoft Windows OS, which does not resemble UNIX very much any more - therefore most of these set up instructions have to do with Apple computers. Windows computers require a great deal more set up and configuration to be made to work like computers running UNIX.

On Mac OS, you will be using the application called Terminal to input commands and run python. Terminal is in the Applications / Utilities folder - drag it to your dock and open it.

On Windows, you will be using a Command Prompt to input commands and run python.

Installing The Code Editor, Sublime Text

Writing your own programs is a lot easier when you use an application specifically designed to make programming easier, much like writing and formatting reports and essays is easier in a word processor application. The editor called Sublime Text is a standard coding tool in the industry, and even though it costs money to use after a trial period, it is well worth trying out. You can go to the Sublime Text official website, visit Downloads, and get the appropriate version for your computer: http://www.sublimetext.com

Mac OS Specific Instructions

Now that you have installed Sublime Text, you will need to edit certain configuration files, known as "dot files", on your system that Apple initially has set to "hidden". In Terminal, run the following commands:

```
defaults write com.apple.finder AppleShowAllFiles -boolean true ; killall Finder
```

You will now be able to see a file in your user home directory called .bash_profile; if it is not there, create it, and add the line:

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

Next you will be installing the Homebrew package manager. Visit the web site http://brew.sh and follow the instructions there to install Homebrew. When you are prompted to install the XCode command line tools, agree to do so and let the install continue. Follow all the instructions that Homebrew displays, then run **`brew update`** and **`brew upgrade`** and **`brew doctor`** and keep following instructions. You will also run **`brew install python`** to create your own local version of Python to program with.

Code Examples (type these in; disregard the first line with filename in parentheses)

(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

    # It is the first roll

        # Check to see if roll is a natural

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

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

        # Check to see if roll is craps

            # If it is 2, 3, or 12 and player bet pass, they lose

            # Player bets don't pass

                # If it is 2 or 3 and player bet don't pass, they win

                # Check to see if roll is a push / tie on a 12

       # Set the point

    # Else

        # Check the roll to see if it matches

            # Check to see if the roll is a 7

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

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

            # The roll 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 yet - keep rolling!
```

```
(craps.py)
import random

from termcolor import colored

die_face = { 1 : unichr(0x2680),
             2 : unichr(0x2681),
             3 : unichr(0x2682),
             4 : unichr(0x2683),
             5 : unichr(0x2684),
             6 : unichr(0x2685) }

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

    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(colored(u"Press Return to roll...", 'cyan'))

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

    result = die_1 + die_2

    if (player_point == 0):
        # It is the first roll
        if ((result == 7) or (result == 11)):
            # Check to see if roll is a natural
            if (player_bet == "p"):
                # If it is 7 or 11 and player bet pass, they win
                print colored("You rolled %d - you win!", 'green') % (result, )
            elif (player_bet == "dp"):
                # If it is 7 or 11 and player bet don't pass, they lose
                print colored("Ouch, you rolled %d and bet \"don't pass\" - you lose. Game Over.", 'red') %
                                (result, )
            break
        elif (result in (2, 3, 12)):
            # Check to see if roll is craps
            if (player_bet == "p"):
                # If it is 2, 3, or 12 and player bet pass, they lose
                print colored("Ouch, you rolled %d and bet \"pass\" - you lose. Game Over.", 'red') %
                                (result, )
            elif (player_bet == "dp"):
                # Player bets don't pass
                if (result in (2, 3)):
```

```
                # If it is 2 or 3 and player bet don't pass, they win
                print colored("You rolled %d - you win!", 'green') % (result, )
            else:
                # Check to see if roll is a push / tie on a 12
                print colored("You rolled %d - game is a push. Play again!", 'cyan') % (result, )
        break
    else:
        # Set the point
        player_point = result
        print u"Your first roll is %s, %s - your point is %d" % (die_face[die_1], die_face[die_2],
                                                                 player_point)
else:
    print u"You rolled %s, %s - result of %d, your point is %d" % (die_face[die_1],
                                                                   die_face[die_2], result, player_point)
    # Check the roll to see if it matches
    if (result == 7):
        # Check to see if the roll is a 7
        if (player_bet == "p"):
            # If it is 7 and player bet pass, they lose
            print colored("Ouch, you rolled %d and bet \"pass\" - you lose. Game Over.", 'red') %
                          (result, )
        elif (player_bet == "dp"):
            # If it is 7 and player bet don't pass, they win
            print colored("You rolled %d - you win!", 'green') % (result, )
        break
    elif (result == player_point):
        # The roll matches the point
        if (player_bet == "p"):
            # The player makes the point and bet pass - a win
            print colored("You rolled %d - you win!", 'green') % (result, )
        elif (player_bet == "dp"):
            # The player makes the point and bet don't pass - they lose
            print colored("Ouch, you rolled %d and bet \"don't pass\" - you lose. Game Over.", 'red') %
                          (result, )
        break
    else:
        # No result matches point or is 7 yet - keep rolling!
        print colored("Keep rolling!", 'green')
```

(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
from termcolor import colored

# Define what a card is; each card has a name, a point value, and a suit
class Card:
    red_suits = [unichr(0x2665), unichr(0x2666)]
    blue_suits = [unichr(0x2663), unichr(0x2660)]

    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):
        if (self.card_suit in self.red_suits):
            color = 'red'
        else:
            color = 'blue'

        return colored(unicode(self.card_name), 'yellow') + colored(self.card_suit, color)

    # 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 it should be 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_unit_test.py)

import random

from game_library.card import *

def print_deck(this_deck):
    int_counter = 0

    for each_card in this_deck:
        int_counter += 1

        print str(int_counter) + ":" + each_card.get_card()


# First create the deck
deck = create_deck()

# Show us the deck - note it will not print “in order”
print_deck(deck)

print "\n"

# Shuffle the deck
random.shuffle(deck)

# Show us the shuffled deck
print_deck(deck)
```

(blackjack.py)

```
import random

from termcolor import colored
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 colored(u"Welcome to blackjack! Highest to 21 wins. Dealing for you...", 'green')

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

print colored(u"You have: " + player_card.get_card(), 'cyan')

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

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

# Deal two cards to the dealer
print colored(u"Now dealing for the dealer...", 'cyan')

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

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

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

print colored(u"Dealer deals another card down.", 'cyan')

# Play the game until the player and 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 colored(u"Player dealt: " + player_card.get_card() + u" , total of " + unicode(player_total),
                      'cyan')

        if (player_total > 21):
            # Check to see if player busts
            print colored("Busted! Game over.", 'red')
            break
        elif (player_total == 21):
            # Check to see if player wins
            print colored("Player has 21!", 'green')
            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 colored(string_dealer, 'cyan')
```

```
        while True:
            # Dealer must hit if points are <= 16 must stand on > 16
            if (dealer_total <= 16):
                # Dealer must hit
                print colored("Dealer hits...", 'cyan')

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

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

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

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

                break

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

Further Reading

Learning Python
Lutz, Mark. O'Reilly Media, Inc., October 2009 (may have later editions)

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. Search engines will usually return results from this site if you search for specific enough keywords, or by copying and pasting the error you receive from the terminal or console into the search box.

Vocabulary

In this section we define a number of terms related to programming. Please let me know if there are any words I use that you don't understand, so I can add them to future versions of this document!

shell

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

variable

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

assignment

The act of storing a value into a variable, using the assignment operator **=**

operator

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

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 different types of values can be stored in variables and values can change their type without causing errors in execution.

execute

To start a program.

Credit & License

The latest version of this packet and all other workshop materials can be found online at Github:

https://github.com/mik3cap/Thinking-Like-A-Programmer-workshop-code-examples/tree/master/Coding-For-Beginners