



# 600.112 IPSE INTRO PROGRAMMING FOR SCIENTISTS & ENGINEERS

<http://www.cs.jhu.edu/~joanne/cs112>

Prof. Joanne Selinski  
Fall 2015

# WEEK 1: 8/31-9/2

- Programming Overview
- Python Overview
- Assignment 0:
  - environment set-up
  - arithmetic experimentation

# WHAT IS A PROGRAM? (STUDENTS)

- set of instructions for a computer
- functions that help the computer understand what to do
- logical steps to accomplish a task
- written in a particular programming language
- writing code that the computer can execute for particular purpose
- code that other humans can follow, edit, upgrade
- takes input, creates output

# WHAT IS A PROGRAM?

- Set of ordered instructions
- Solves a problem
- Computer can execute
- Unambiguous
- Terminating
- Takes input, processes, creates output

# PROGRAM STYLES

## ○ GUI

- graphical user input based, event based
- mostly what you're probably used to
- example: Mac and Windows based OS

## ○ Text-based

- user types input and gets output interactively and sequentially as the program executes
- mostly the type we will create
- example: Linux (Unix) and MS-DOS based OS

## ○ Batch processing

- input and output data is in files with very little user interaction

# PROGRAMMING PHASES

# PROGRAMMING PHASES

1. Define & analyze: know requirements, clarify inputs & outputs, get any formulas, create a sample run/use case scenarios with real data and results

# PROGRAMMING PHASES

1. Define & analyze: know requirements, clarify inputs & outputs, get any formulas, create a sample run/use case scenarios with real data and results
2. Design: overall program structure, algorithms in pseudocode

# PROGRAMMING PHASES

1. Define & analyze: know requirements, clarify inputs & outputs, get any formulas, create a sample run/use case scenarios with real data and results
2. Design: overall program structure, algorithms in pseudocode
3. Write program code: type, compile, debug, run

# PROGRAMMING PHASES

1. Define & analyze: know requirements, clarify inputs & outputs, get any formulas, create a sample run/use case scenarios with real data and results
2. Design: overall program structure, algorithms in pseudocode
3. Write program code: type, compile, debug, run
4. Testing – find errors, go to step 3

# PROGRAMMING PHASES

1. Define & analyze: know requirements, clarify inputs & outputs, get any formulas, create a sample run/use case scenarios with real data and results
2. Design: overall program structure, algorithms in pseudocode
3. Write program code: type, compile, debug, run
4. Testing – find errors, go to step 3
5. Documentation – for the user

# PROGRAMMING PHASES

1. Define & analyze: know requirements, clarify inputs & outputs, get any formulas, create a sample run/use case scenarios with real data and results
2. Design: overall program structure, algorithms in pseudocode
3. Write program code: type, compile, debug, run
4. Testing – find errors, go to step 3
5. Documentation – for the user
6. Maintain/upgrade

# ALGORITHMS

- Put on a hoodie
- Calculate an average
- Compute homework point sum
- Calculate your GPA

# PSEUDOCODE DRAMATIZATION

- volunteer readers
- volunteer actors

## 3 TYPES OF PROGRAM CONTROL

- Sequential statement execution (default)
  - Decision statements
  - Repetition statements (loops)
- 
- All general purpose programming languages must have ways to do these 3 things!

## ALGORITHM – AVERAGE 3 NUMBERS

- prompt for 3 numbers
- read 3 numbers, store as num1, num2, num3
- add num1, num2 and num3, store as result
- divide result by 3 (restore new value in result)
- display "average is", result

## ALGORITHM – HOMEWORK POINT SUM

- Get homework point values for one student, calculate the sum, "fail" if  $< 150$
- Ideas for what the program interaction might be?

## APPROACHES – HOMEWORK POINT SUM

- Get homework point values for one student, calculate the sum, "fail" if  $< 150$
- A) Ask the user how many homework grades there are at the start.
- B) Ask for the grades and have the user enter a -1 or "stop" at the end (some sentinel value).
- C) Ask the user if they have more data after every value is input.

# HOMEWORK POINT SUM – SAMPLE RUN – VERSION A

Welcome to the homework point sum program!

Enter # of homework grades: 4

Enter grade 1) 40

Enter grade 2) 50

Enter grade 3) 38

Enter grade 4) 45

Homework sum is 173.

# HOMEWORK POINT SUM – SAMPLE RUN – VERSION B

Welcome to the homework point sum program!

Enter grades, STOP to end:

30 30 30 30 STOP

Homework sum is 120. Fail!

# HOMEWORK POINT SUM – VERSION A

## PSEUDOCODE

- print welcome message
- prompt for number of hw grades
- read numhw
- initialize sum to 0
- repeat numhw times
  - prompt for grade (using grade #)
  - read grade
  - add grade to sum
- display "homework sum is", sum
- if  $\text{sum} < 150$ , display "Fail"

# HOMework POINT SUM – VERSION C

## PSEUDOCODE

## 3 TYPES OF PROGRAM CONTROL

- Sequential statement execution (default)
  - Decision statements
  - Repetition statements (loops)
- 
- All general purpose programming languages must have ways to do these 3 things!

# PROGRAMMING LANGUAGES

- 4) Pseudocode – english-like program
- 3) High-Level (just a sampling)
  - scripting: python, Javascript, Ruby
  - object-oriented: Java, C++, C#
  - number crunching: Matlab, R, FORTRAN, Mathematica
  - procedural: C, Pascal, BASIC
- 2) Assembly, Java Bytecode
- 1) Binary = machine = executable code

# SAMPLE PROGRAMS – 3 LANGUAGES

```
#include <iostream>
using namespace std;
int main() {
    string name;
    cout << "What is your name? ";
    cin >> name;
    cout << "Greetings " << name << endl;
}
```

```
name = raw_input("What is your name? ")
print "Hello " + name
```

```
import java.util.Scanner;
public class hola {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        System.out.print("What is your name? ");
        String name = in.nextLine();
        System.out.println("Hola " + name);
    } }
```

# PROGRAM TRANSLATION

- from high level to lower level
- Compilers
  - take program source code, produce a new file which is a lower level (usually executable)
  - examples: C, FORTRAN, C++, Java (to bytecode)
- Interpreters
  - translate program source code while executing the program
  - examples: BASIC, Python, Java Bytecode

# TRANSLATE & RUN SAMPLE PROGRAMS

- `g++ greetings.cpp`

- `./a.out`

What is your name? Joanne

Greetings Joanne

- `javac hola.java`

- `java hola`

What is your name? Joanne

Hola Joanne

- `python hello.py`

What is your name? Joanne

Hello Joanne

# WHY PYTHON?

- Easy to learn
- Easy to experiment
- Similar to elements of MATLAB, C++, JavaScript
- Lots of industry and research use (Google)
- Lots of useful libraries
  - plotting, data analysis
  - interactive graphics, visualization
  - read/write data in various formats

# HOW PYTHON?

- Interactive shell for live code execution (good for experimenting & learning)
- 'Idle' software for creating and running programmings (good for making modules for reuse too)
- Install and import lots of library modules to make it easy to do cool stuff
- VirtualBox – so we are all using a consistent platform and the same libraries
- (insert demo here)

## ENVIRONMENT SET-UP (LAB)

- Let's work together to make sure everyone can run the python interpreter and Idle to create python programs...
- To be continued in the lab sections this week, or go to office hours if you need help

## WEEK 1.5: 9/9

- Diving in: GPA example

## GPA EXAMPLE

- Let's start the process of writing a program to compute a GPA for any set of courses.
- Problem analysis & design (pseudocode):  
[lects/gpa.txt](#)
- Python code: [lects/gpa.py](#)

## WEEK 2: 9/14-9/16

- Turtle Graphics
- Functions, Loops, Ranges
- Python Language Details
- Assignment 1:
  - making shapes
  - drawing function curves

# TURTLE GRAPHICS - OVERVIEW

- ✎ The **turtle** module in Python is a fun and simple way to create graphics.
- ✎ A turtle object can draw lines of various sizes and lengths by dragging its "tail" when the pen is down, or move to new spots with the pen up
- ✎ The turtle lives in a graphics window and its location is indicated by (x,y) coordinates, where (0,0) is the center of the window
- ✎ Turtles also have orientation, which is an angle relative to the x-axis, facing in the positive direction, called its heading.
- ✎ **import** turtle to use
- ✎ turtle.setup() to create window
- ✎ turtle.done() at end

## TURTLE GRAPHICS – USEFUL METHODS

- ✎ `import turtle` to use
- ✎ `turtle.setup()` to create window
- ✎ `turtle.done()` at end
- ✎ `turtle.down()` and `turtle.up()` for pen control
- ✎ `turtle.left(angle)` and `turtle.right(angle)` to change direction, where angle is degrees
- ✎ `turtle.forward(length)` and `turtle.backward(length)` to move length steps
- ✎ `turtle.goto(x,y)` to move to a specific location
- ✎ `turtle.color("red")` to change the pen color
- ✎ look at the on-line documentation for more:  
(<http://docs.python.org/2/library/turtle.html>)

# RANGES

- ✎ Python has a built-in range function to create a sequence of values.
- ✎ Simple version: `range(value)` creates a sequence of integers from 0 (inclusive) to value (exclusive):
  - ✎ `[0, 1, ..., value-2, value-1]`
- ✎ Full version: `range(start, stop, step)` creates a sequence of integers starting at start (inclusive), stopping at stop (exclusive), in increments of size step
  - ✎ `range(10, 100, 5) => [10, 15, 20, ..., 90, 95]`
  - ✎ `range(0, -90, -20) => [0, -20, -40, -60, -80]`

# LOOPING THROUGH RANGES

- We generally use a `for` loop with a range to repeat a block of code.
  - General form:
- ```
for var in sequence:  
    do stuff
```
- Here, `sequence` is a list of values, such as that generated by a range, and `var` will take on each value in the sequence, one at a time.
  - We can also loop through a range without `var` to simply repeat steps a certain number of times:

```
for _ in range(100):  
    do stuff 100 times
```

# FUNCTIONS

There are two elements to working with functions.

✎ defining the function statements (what it does)

```
def funcName((opt)parameters):  
    statement1  
    statement2  
    return (optional) value
```

✎ calling the function with actual arguments to make it execute

```
funcName((matching)arguments)
```

✎ parameters and return statements are optional

✎ return statements exit the function immediately

## IDLE HELP

- to discover more methods for a particular object type: after creating a particular object named `var`, type `var .` and wait to see what methods pop up in the hint menu that appears
- if you know the name of a function, in the interpreter type `help(name)` to get information about it

# PYTHON LANGUAGE DETAILS




- Python Basics
  - language elements
  - data types
  - operators
  - expressions

# PYTHON PROGRAM COMPONENTS

- ✎ data values: numbers, strings, lists (sequences)
- ✎ variables to name our values
- ✎ operators to manipulate values
- ✎ statements to execute (assignments, function calls, decisions, loops)
- ✎ functions to bundle statements into subroutines
- ✎ classes to bundle data and methods (functions) to manipulate more complex objects
- ✎ modules to bundle related components together
- ✎ comments to explain the code

# PYTHON LANGUAGE ELEMENTS




## Reserved words

-  core to the language
-  can only be used as intended
-  examples: `and, as, def, else, elif, False, for, if, import, in, is, not, or, pass, print, return, True, while`

## Symbols (operators)

-  have predefined meanings based on context




## Identifiers

-  used for names of: variables, functions, classes
-  we make these up
-  contain: letters, `_`, digits but not as first character


## Literal values: numbers, strings, lists

# (MORE) PYTHON LANGUAGE ELEMENTS




## Comments

-  not executed – documentation only
-  single line style: ignore from # to end of line
-  block style for docstrings: " or """ to start and end comment (must match), can extend over several lines of code



## Case sensitive

-  must be consistent in use of capitalization

## White space

-  indentation is important!
-  use 4 spaces for each new block level
-  use blank lines to separate logical units

## Line lengths

-  limited to 79 characters
-  use \ to continue a statement to the next line

# NUMBER TYPES & ARITHMETIC OPERATORS

✎ int – integer whole numbers: 15 -2453 291039

✎ long – big whole numbers: 15L

✎ float – floating point numbers: 14.203 -234E10

✎ (complex – imaginary numbers: 3j )

✎ \*\* exponentiation

✎ \* multiplication

✎ / (float) division, // truncated division, % mod

✎ + addition

✎ - subtraction

## DIVISION RESULTS – PYTHON2 VS PYTHON3

Python 2.7.3

```
>>> 23 / 5 => 4
```

```
>>> 23.0 / 5 => 4.6
```

Python 3.3.2

```
>>> 23 / 5 => 4.6
```

```
>>> 23.0 / 5 => 4.6
```

Same on both:

```
>>> 23 // 5 => 4
```

```
>>> 23.0 // 5 => 4.0
```

```
>>> 23 % 5 => 3
```

```
>>> 23.0 % 5 => 3.0
```

## OTHER DATA TYPES

✎ bool – boolean values

✎ True or False values only

✎ sequences

✎ str – strings are sequences of characters

✎ single quote or double quote delimited

✎ 'this is a string'

✎ "this is also a string"

✎ list – a sequence of values

✎ comma separated sequences of values in [ ]

✎ [1, 2, 3, 4]

✎ ['a', 'bcd', 'e']

✎ [1, "aa", 2, "BBB", 3]

# CHARACTER REPRESENTATION

- ✎ each character is assigned an integer code
- ✎ full set is Unicode System
- ✎ extension of original ASCII system
- ✎ 'A' to 'Z' have consecutive codes (65-90)
- ✎ 'a' to 'z' have consecutive codes (97-122)
- ✎ '0' to '9' have consecutive codes (48-57)

## DATA TYPE CONVERSIONS - Casting

✎ Explicit conversions – type these in the Python interpreter and see what you get:

✎ `int(24.645)` - 24

✎ `float(23 // 5)` - 4.0

✎ `int("1324")` - 1324

✎ `str(5+9)` - "14"

✎ `ord('a')` - 97 (Unicode value of a single character)

✎ `chr(98)` - 'b' (Character corresponding to valid Unicode)

# ASSIGNMENT 1

- Let's work on part 2 together...

## WEEK 3: 9/21-23

- Decisions
  - Boolean Expressions
  - Simple I/O
  - Loops
  - Docstrings
- 
- Assignment 2 - Part 2

# DECISIONS, DECISIONS

- ✎ Decision statements allow us to make a choice based on the result of a test or condition
- ✎ Pseudocode example:
  - if age greater than or equal to 16
  - then get driving permit
  - otherwise keep walking
- ✎ Python has three ways of using the built-in decision statement:
  - ✎ **if** – one way decision
  - ✎ **if/else** – two way decision
  - ✎ **if/elif** – nested series of decisions

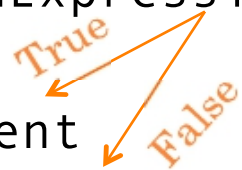
## ONE-WAY IF

✎ General format:

```
if booleanExpression:
```

```
    statement
```

```
nextStatement
```



✎ If the boolean expression is True, the statement gets executed.

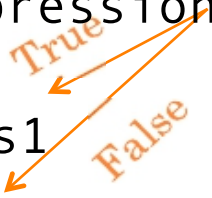
✎ If the boolean expression is False, the statement is skipped.

✎ Program execution continues with whatever follows (nextStatement).

## TWO-WAY IF/ELSE

✎ General format:

```
if booleanExpression:  
    statements1  
else:  
    statements2  
nextStatement
```



- ✎ If the boolean expression is True, only statements1 get executed.
- ✎ If the boolean expression is False, only statements2 get executed.
- ✎ Program execution continues with whatever follows (nextStatement).

## NESTED IF/ELIF/ELSE

✎ General format:

```
if booleanExpression1:
    statements1
elif booleanExpression2:
    statements2
elif booleanExpression3:
    statements3
else:
    statementsLast
nextStatement
```

✎ You can have as many elif parts as you need.

# BOOLEAN EXPRESSIONS

We form boolean expressions with

✎ comparison operators, for numbers and strings

✎ `<` `>` `<=` `>=`

✎ `==` (equals)

✎ `!=` (not equals)

✎ membership operators, for sequences

✎ `in`

✎ `not in`

✎ logical operators, for combining boolean values


✎ `not`

✎ `and`

✎ `or`

# LOGICAL OPERATORS


## not

  $\text{not True} \Rightarrow \text{False}$ ,  $\text{not False} \Rightarrow \text{True}$

## and

  $\text{True and True} \Rightarrow \text{True}$

  $\text{True and False} \Rightarrow \text{False}$

  $\text{False and True} \Rightarrow \text{False}$

  $\text{False and False} \Rightarrow \text{False}$

## or

  $\text{True or True} \Rightarrow \text{True}$

  $\text{True or False} \Rightarrow \text{True}$

  $\text{False or True} \Rightarrow \text{True}$

  $\text{False or False} \Rightarrow \text{False}$

## DEMORGAN'S LAWS

✎  $\text{not } (A \text{ and } B) == \text{not } A \text{ or not } B$

✎  $\text{not } (\text{raining and cold}) == \text{not raining or not cold}$

✎  $\text{not } (\text{let} \geq 'A' \text{ and let} \leq 'Z') == \text{let} < 'A' \text{ or let} > 'Z'$

✎  $\text{not } (A \text{ or } B) == \text{not } A \text{ and not } B$

✎  $\text{not } (\text{Tuesday or IPSE}) == \text{not Tuesday and not IPSE}$

✎  $\text{not } (\text{ans} == 'y' \text{ or ans} == 'Y') == \text{ans} != 'y' \text{ and ans} != 'Y'$

# OPERATORS HAVE PRECEDENCE

✎ () do inside parentheses first

✎ - negation

✎ \*\* exponentiation

✎ \* multiplication, / division, // div, % mod

✎ + addition, - subtraction, + concatenation

✎ < > <= >=

✎ == != in not in

✎ not


✎ and

✎ or


✎ = assignments, +=, -=, \*=, /=, etc.

# COMMON MODULES

## Math


 <http://docs.python.org/2.7/library/math.html?highlight=math#math>

## Random

 <http://docs.python.org/2.7/library/random.html?highlight=random#random>

# MATH MODULE

 `import math`

 contains lots of useful functions – rounding, exponents and logs, trig functions, etc.


 most methods return float data values


 examples – type these into the Python shell:


 `math.pow(12, 3.5)`

 `math.sqrt(243)`

 `math.round(24.345)`

 `math.floor(24.6948)` (round down)

 `math.ceil(24.345)` (round up)

 useful constants: `math.pi`, `math.e`

# RANDOM

✎ `import random`

✎ used to generate "random" data

✎ based on pseudorandom sequence

✎ uses current time as the seed (sequence start)

✎ common methods:

✎ `randint(a, b)` – random integer in range [a, b]

✎ `random()` – returns float in range [0.0, 1.0)

✎ `randrange(start, stop, step)` – random element from the range specified

✎ we can massage data into customized forms with transformations in our code

✎ examples: `randUpper`, `randBoolean`

## RANDOM EXAMPLES

```
def randUpper():  
    ch = random.randint(ord('A'), ord('Z'))  
    return chr(ch)
```

```
def randBool():  
    val = random.randint(0,1)  
    if val == 1:  
        return True  
    else:  
        return False  
    # return val == 1
```

# SIMPLE INPUT & OUTPUT

- User input can be gotten with two methods:
  - `raw_input("prompt")` – will display the "prompt" and get the user input, returning it as a string
    - we can then use our casting operators to explicitly convert this into other types, such int or float
  - `input("prompt")` – will display the "prompt" and read input, implicitly evaluating it and returning the result
- Textual output is generated by the print function and will appear in the interpreter window
  - `print item1, "some string"` – will print the contents of variable `item1` followed by a space and then `some string` finally moving the cursor to the next line (this function differs in Python3)

# LOOPS IN GENERAL

✎ Loops allow us to repeat one or more statements based on the result of a test or condition

✎ Loop control mechanisms:

✎ counter: repeat a certain number of times

✎ sentinel: repeat until a value or event occurs

✎ iterator: repeat for every value in a collection

✎ Pseudocode examples:

repeat 10 times (counter controlled)

repeat until you run out of input (sentinel controlled)

repeat for every number in a set (iterator controlled)

# LOOPS IN PYTHON

✎ Python has two types of built-in repetition statements:

✎ while


✎ for

# WHILE LOOP

✎ Good for counter or sentinel control

✎ General form:

```
while booleanExpression:  
    statements1  
nextStatement
```



✎ For as long as the boolean expression is True, the block of statements1 gets repeatedly executed

✎ Whenever the boolean expression is False, the loop ends and the control continues with nextStatement

# FOR LOOP

✎ Used mostly for iterator or counter control

✎ General form:

```
for var in sequence:  
    statements1  
nextStatement
```

✎ for as long as there are values to be processed, the block of statements1 will be repeated

✎ when it runs out of values to process, the loop ends and the control continues with nextStatement

## LOOP EXERCISES

- write a "for" loop to implement version A of the homework sum problem: ask the user how many values there are, then input and add them up (posted as hwsums.py)
- write a "while" loop to implement version B of the homework sum problem: tell the user to enter values one per line, using QUIT at the end (posted as hwsums.py)

# DOCSTRINGS

- We use block comments, called docstrings in Python, to provide user documentation for our programs, modules and functions.
- Use `"` or `"""` to start and end each docstring.
- Include all information pertinent to using the unit being documented.
- For functions be sure to explain what the parameters represent, and what results or returned or output.
- Do this for every function and every program you write from now on.

## ASSIGNMENT 2 – BROWNIAN MOTION

- Let's work on part 2 of assignment 2 together...

## WEEK 4: 9/28-30

- Sequences
  - Strings
  - File I/O
  - Testing
- 
- Assignment 3: Genome Sequences

# SEQUENCES

- Sequences are collections of data
  - str – strings are sequences of characters
  - list – sequence of values
    - comma separated, enclosed in [ ]
    - can be all of same type, or different types
    - Python's version of an array
- There are built-in operations that can be performed on sequences through operators, functions or methods.
- Individual data values are accessed by integer indices, enclosed in [ ]
  - ranging from 0 to length-1, left to right
  - negative indices start from the right end, and go from -1 (last element) to -length (first element)

## SEQUENCE OPERATORS

- ✎ `seq[ie]` – can be used to access an element of a sequence, where *ie* is a valid integer expression
- ✎ `seq1 + seq2` – can be used to join (concatenate) two sequences `seq1` and `seq2` together, creating a new sequence `result`
- ✎ `seq * ie` – will create a new sequence containing *ie* concatenated copies of the sequence `seq`, where *ie* is a positive integer expression
- ✎ `val in seq` – **True** when `val` is in the sequence, **False** otherwise
- ✎ `len(seq)` – gives the number of items in the sequence

## MORE SEQUENCE OPERATIONS

- `seq[i1:i2]` – get a sub-sequence
  - slices a sequence by returning a subsequence containing the items starting at index *i1*, up to but not including the item at index *i2*
  - if *i1* is omitted, the default is 0 (the start)
  - if *i2* is omitted, the default is the end
  - the original sequence is not modified
- `for val in seq:`
  - iterates over every value in the sequence `seq`
- `for i in range(len(seq)):`
  - iterates over every index that's valid for `seq`
  - can use to change each `seq[i]` for example

# SEQUENCE METHODS

- ✎ methods are functions that are applied to sequences in an object oriented way
- ✎ `seq.method_name([args])` is the general form
- ✎ some examples:
  - ✎ `seq.count(item)` – returns the number of occurrences of a specific item
  - ✎ `seq.index(item)` – returns the index of the first occurrence of item if its in seq, error otherwise
  - ✎ `seq.remove(item)` – removes the first occurrence of item if its in seq
  - ✎ `seq.insert(index, item)` – inserts item at position index in the seq

## Recall: RANGES

- ✎ Python has a built-in range function to create a sequence of values.
- ✎ Simple version: `range(value)` creates a sequence of integers from 0 (inclusive) to value (exclusive):
  - ✎ `[0, 1, ..., value-2, value-1]`
- ✎ Full version: `range(start, stop, step)` creates a sequence of integers starting at start (inclusive), stopping at stop (exclusive), in increments of size step
  - ✎ `range(10, 100, 5) => [10, 15, 20, 25, ..., 90, 95]`
  - ✎ `range(0, -90, -20) => [0, -20, -40, -60, -80]`

# STRING OPERATIONS

- ✎ Strings have special operations (methods) that do not apply to all types of sequences.
- ✎ You can look them up here:
  - ✎ <http://www.python.org/doc/current/library/stdtypes.html#string-methods>
- ✎ A few examples:
  - ✎ `cstring.find(item, index)` – starting at position *index* in *cstring* (or 0 if *index* is not specified, searches for and returns the (starting) index of the first occurrence of *item*, or -1 if not found
  - ✎ `cstring.upper()` – returns an uppercase version of *cstring*
  - ✎ `cstring.rjust(w)` – returns *cstring* right justified in a field of *w* characters total, padded with spaces
  - ✎ `cstring.split(item)` – returns a sequence of substrings using *item* as the delimiter (where to split). By default whitespace will be used as the delimiter if *item* is not specified.

## RECALL: IDLE HELP

- to discover more methods for a particular object type: after creating a particular object named `var`, type `var .` and wait to see what methods pop up in the hint menu that appears
- if you know the name of a function, in the interpreter type `help(name)` to get information about it

# USING STRINGS & SEQUENCES

- Idle interpreter demo of common operations:
  - sequence operations, indices, slicing
  - string methods
- Examples are posted on the course website (see schedule).
- Let's write a pig latin translator! (code posted)

## FILES FOR I/O

- ✎ We can use plain text files for both input and output – reading and writing strings only.
- ✎ We use the open function to initialize an external file with the filename (string) and optional mode:
  - ✎ "r" (default) – open file for reading, must already exist
  - ✎ "w" – create or overwrite existing file for writing
  - ✎ "a" – open existing file to append to the end
  - ✎ `file = open("somefile", "w")`
- ✎ When finished, we close the file:
  - ✎ `file.close()`

## READING FROM FILES

- ✎ Files have iterators so that we can use our common for loop to get every line in the file:  
✎ `for line in file:`
- ✎ It is good practice to strip the line of surrounding whitespace when you read it.
- ✎ We can use the string split function to break it into tokens of information – remember this will create a sequence of strings. You might need to convert them to other types if you intend to use them as numbers.

## WRITING TO FILES

- ✎ We use the write function, which must be given a string value.
  - ✎ `file.write("some string")`
- ✎ Write does not include any spacing or newline characters automatically, so we have to add them ourselves.

# SPECIAL CHARACTERS

- We need a special way of referring to certain symbols or keys, such as tabs and enter/return to go to the next line.
- These are called escape characters, and are preceded by a backslash (\).
  - \n – end of line (enter/return)
  - \t – tab
  - \\ - backslash (since one is used to escape other characters)
  - \b – bell
- Enclose them in single or double quotes to use in Python.

## FILE I/O EXAMPLE

- This piece of code reads from a file and writes every token to a new file, one per line.

```
infile = open("in.txt")
outfile = open("out.txt", "w")
for line in infile:
    tokens = line.strip().split()
    for tok in tokens:
        outfile.write(tok + '\n')
infile.close()
outfile.close()
```

- Note the use of nested loops!

## ASSIGNMENT 3 – GENOME SEQUENCES

- Let's work on part 2 of assignment 3 together...

## WEEK 5: 10/5-7

- Modules
- Function Scoping
- Doctest – testing functions
- Revised: Pig-Latin translator
  
- Functions as parameters
- Assertions
- More sequences: nested lists
  
- Assignment 4

# MODULES

- ✎ A module is simply a collection of function definitions, but no executable program statements.
- ✎ We can write and use our own modules for the ultimate in code reuse.
- ✎ Name your module file something.py as you would any program.
- ✎ In order to use it in a program or the interpreter, you must first import it (note we don't say .py):
  - ✎ `import something`
- ✎ The module needs to reside in the same folder as the program using it.

## FUNCTIONS – VARIABLE SCOPING

- Variables "live" within the blocks in which they are initialized, including
  - function parameters
  - variables controlling and within for loops
- Most variables should be local to the functions in which they are used.
- Variables with the same name but in different functions or blocks are completely unrelated to each other.
- See examples in [scope.py](#)

## GLOBAL VARIABLES

- Global variables may be created outside of any functions.
- You can reference (read access) global variables within any block as long as there isn't a local variable with a conflicting name.
- In order to assign a value to a global variable within a block, you must first declare it so it is not presumed to be a new variable local to that block:

```
global gvar  
gvar = value
```

- See examples in [scope.py](#)

# TESTING

- White-box Testing: each possible path in a program (all possible decision cases) should be tested.
- Black-box Testing: test problem requirements, ignoring code
- Boundary cases (the = part of  $\leq$  or  $\geq$ ) should be tested.
- Valid values should be tested.
- Invalid values should be tested.
- Regression Testing: when rewriting and updating code, be sure to re-test cases that worked before the "upgrade".

# DOCTEST FOR UNIT TESTING

- Unit Testing: individually test each method with it's own mini driver program – incorporate white-box testing.
- Within our docstrings for each function that we write, we can include unit tests to document the expected function behaviour for various inputs (parameters).
- The format of the tests is how you would call the function in the interpreter, and the corresponding result that would be displayed.
- The goal is to cover every possible situation that the function may need to handle.
- There is a module for python called "doctest" that we can use to run the tests.

## DOCTEST EXAMPLE

```
def add(param1, param2):  
    """  
    This function returns the sum of two  
    values.  
    >>> add(13, 10)  
    23  
    >>> add(-2.3, 0)  
    -2.3  
    >>> round(add(32.5, 24.5), 1)  
    57.0  
    """  
    return param1 + param2
```

## USING DOCTEST IN IDLE

- ✎ IDLE does not normally have support for doctest.
- ✎ A former TA wrote a plug-in that you can download – see instructions on Piazza.
- ✎ Once installed, the Run menu will have a "Doc Test" option – just click
- ✎ A new window will open with the results (pass or fail) for each test in each function.
- ✎ This works best if your functions are in a module, not an actual program.
- ✎ Some problems exist if using the plug-in with interactive program input.


## EXAMPLE: PIG-LATIN TRANSLATOR

- Let's update our pig-latin translator!
  - understanding function scoping of variables
  - using docstrings and doctests
- Solution is posted on the website


## USING UNIX – BASIC COMMANDS

- ✎ To get started using unix (linux) in VirtualBox, start with the Accessories menu and open LXTerminal.
- ✎ There are some basic commands to help you move around:
  - ✎ `cd somefolder` (change directory to somefolder)
  - ✎ `cd ..` (go back to the previous (enclosing) directory)
  - ✎ `ls` (list the directory contents)
  - ✎ `cat somefile` (display (concatenate) somefile)
  - ✎ `clear` (clear the window)
- ✎ For a much fuller introduction, read:
  - ✎ <http://www.cs.jhu.edu/~joanne/unix.html>

# USING UNIX FOR PYTHON


 To run a program:


 navigate to the directory that contains the file

 `> python somefile.py`

 To use the interpreter:

 `> python`

 To create a python file use any text editor – try Emacs:

 `> emacs somefile.py`

 You can also launch Emacs from the Accessories or Programming menus in VirtualBox

## USING DOCTEST IN UNIX

- ✎ We simply navigate to the directory where the python file resides. (use `cd` – change directory)
- ✎ We run python with the `-m` (module) option:  
🔗 `python -m doctest myfile.py`
- ✎ The results (pass or fail) for each test in each function will be displayed in the same unix window.
- ✎ If `myfile.py` is a program (not just a module) this will also run the program.

# ASSERTIONS

- ✎ Often a function will require certain conditions or assumptions are true about the parameter values. In programming lingo, we call these *pre-conditions*.
- ✎ Python has a mechanism for checking whether pre-conditions are met before proceeding with the statements in a function:
  - ✎ `assert booleanExpression`
- ✎ If the `booleanExpression` is `True`, execution proceeds with the next statement.
- ✎ If the `booleanExpression` is `False`, an error message will be printed and execution halts.
- ✎ For now, only use `assert` if you want a failed assertion to stop the program execution. If not, just do a simple boolean test, print an error message, and return [a dummy value if necessary] from the function instead.
- ✎ See `assert.txt` for examples.

## FUNCTION NAMES AS PARAMETERS

- ✎ We can pass functions as parameters to other functions simply by using their names.
- ✎ This allows us to customize a function by letting it call different other functions based on the parameter.
- ✎ See `functionParameters.txt` for examples.

# STRINGS, LISTS & TUPLES

- ✎ These are all types of sequences.
- ✎ Strings are sequences of characters only, and are immutable – individual characters cannot be changed, only accessed with indices.
- ✎ Lists are mutable – that means we can change individual elements.
- ✎ Tuples can contain any types of value (like lists), but they are immutable too.
  - ✎ represent with () or nothing, not [] to differentiate from lists:

```
tup = (1, 3, 'two', 4.5)
trip = 5, 10, 15
```
  - ✎ access elements with []:

```
print tup[1]    => 3
print trip[2]   => 15
```

## NESTED LISTS

- ✎ We can create lists of lists, called nested lists.
- ✎ One level of nesting can be used to represent tabular data (2 dimensional array or matrix).
- ✎ Create nested list with 2 rows, 3 columns each:  
✎ `table = [[1, 2, 4], [3, 6, 10]]`
- ✎ Access or change whole rows:  
✎ `print table[0] => [1, 2, 4]`  
✎ `table[0] = [-2, -3, -5]`
- ✎ Access or change individual elements:  
✎ `print table[1][1] => 6`  
✎ `table[1][0] = 20`
- ✎ There isn't a simple way to access a whole column.

## MORE NESTED LISTS

- ✎ Sublists in a nested list may have different lengths:
  - ✎ `nestList = [[1, 2], [1, 4, 6], [3, 5, 10, 16]]`
  - ✎ `nestList[0][3]` doesn't exist, but `nestList[2][3]` does
- ✎ Nesting is not limited to 2 dimensions – you can have lists of lists of lists of lists, lists of lists of strings, lists of tuples of lists of strings, etc.
- ✎ See `sequences.txt` for examples of strings, lists, tuples, and nested lists.

## ASSIGNMENT 4 - ECHOCARDIOGRAMS

- Let's work on Part 2 of assignment 4 now...

## WEEK 6: 10/12-15 (3 MEETINGS)

- Quiz 1! on topics through project 3 (strings & sequences)
- Pep8 Style Checker
- List Comprehensions
- Matrices (using nested lists & loops)
- Python Sets
- Recursion basics
- Assignment 5

# PEP8 STYLE CHECKER

✎ <http://www.python.org/dev/peps/pep-0008/>

- ✎ The purpose is to insure coding style consistency among all those developing python modules and programs.
- ✎ Mostly the rules consist of spacing guidelines and naming guidelines.
- ✎ To download: use Synaptic Package Manager
- ✎ To run in IDLE: see pep8/doctest plug-in instructions on Piazza, select "Style Check" from the Run menu.
- ✎ To run in LXTerminal: just type "pep8 myfile.py" at the prompt.

## LIST COMPREHENSIONS - EXAMPLES

- ✎ We can do some funky things to create lists, besides using straightforward ranges.
- ✎ These types of initializations are called list comprehensions.
- ✎ Some cool examples:

```
>>> [2**i for i in range(10)]  
[1, 2, 4, 8, 16, 32, 64, 128, 256, 512]  
>>> words = ["one", "two", "three", "four"]  
>>> [ w[0].upper() for w in words ]  
['O', 'T', 'T', 'F']  
>>> [ 3 * x + 2 for x in range(10, 100, 3) if x % 5 == 0 ]  
[32, 77, 122, 167, 212, 257]
```

## LIST COMPREHENSIONS IN GENERAL

```
listB = [expr(i) for i in listA if f(i)]
```

is equivalent to

```
listB = []  
for i in listA:  
    if f(i):  
        listB.append(expr(i))
```

`f(i)` is a filter that returns a **True/False** value used to select certain elements from `listA`

`expr(i)` is any expression, usually using the value `i`, that creates the values that go into `listB`

Source: <http://www.pythonforbeginners.com/lists/list-comprehensions-in-python/>

# WORKING WITH MATRICES

- ✎ 2-dimensional nested lists represent matrices

- ✎ Common operations:

  - ✎ create an initialized matrix of a specific size

  - ✎ create an identity matrix of a specific size

  - ✎ print a matrix in tabular format

  - ✎ add two matrices

  - ✎ apply a function to a column (or row)

  - ✎ multiply two matrices

- ✎ See `matrices.py` for python code to do some of these things.

## VALIDATING DATA TYPES

✎ Sometimes it is useful to check the type of data associated with a particular variable name.

✎ We can do this using the `type()` function:

```
>>> var = 24
>>> type(var)
<type 'int'>
>>> var = "24"
>>> type(var)
<type 'str'>
>>> if type(var) is int:
...     print 'var is int'
... else:
...     print 'var is NOT int'
...
var is NOT int
```

# RECURSION

- ✎ solve problem by breaking into smaller pieces
- ✎ solve each piece with same strategy
- ✎ put pieces together for original solution
- ✎ a way of doing repetition

## ✎ Recursive method

- ✎ call(s) itself – recursive case(s)
- ✎ base case(s) – does not call itself

## ✎ Examples: recurse.py

# PYTHON SETS

- Data type to hold an *unordered* collection of *unique* immutable values.
- No indexing, slicing, etc.
- (More like a dictionary than a list.)
- Sets can be mutable (default) or made immutable.
- <http://docs.python.org/2/library/sets.html>
- Examples in setScript.txt

# SET OPERATIONS

- Initialize by applying the set function to a collection:

```
set1 = set([1, 3, 5, 7])
```

- Many operations on sets have operator forms and method forms:

```
set1.union(set2) ~ set1 | set2  
set1.difference(set2) ~ set1 - set2  
s1.intersection(s2) ~ s1 & s2  
s1.issubset(s2) ~ s1 <= s2
```

- Other operations only have method forms:

```
set1.add(x)  
set1.remove(x) or set1.discard(x)
```

## ASSIGNMENT 5 – PART 2

- let's do some sudoku solving!

## WEEK 7: 10/19 & 10/21

- Bringing vectors & matrices to life:
  - Plotting with pyplot from Python's matplotlib
  - Numpy for fast matrix operations
- Assignment 6

# PLOTTING IN PYTHON

- The **matplotlib** library contains many modules with features and functions for 2D plotting.
- Some of these are common to MATLAB.
- [matplotlib.org](http://matplotlib.org) contains main examples and links to tutorials.
- We'll be using the **pyplot** module from this library.
- Note: Install **matplotlib** with the synaptic package manager if you haven't yet.

## IMPORT OPTIONS

- When we import just a particular module from a larger library, we need to use both names:
  - `import matplotlib.pyplot`
- However, it gets rather inconvenient to use `matplotlib.pyplot` to call every function we may want to use. Instead python provides an aliasing option when you import:
  - `import matplotlib.pyplot as plt`
- This allows us to refer to the module simply as `plt`
- You can use any alias name, but keep it fairly descriptive particularly in longer programs since it needs to be distinct from variable names.

# MATPLOTLIB.PYPLOTTING COMMON FUNCTIONS

- Here are simplified usages of common functions:
  - `plot(v)` – plot a line based on the values in vector `v` – the x values of the points on the line are the indices and the y values are the values in the vector; you can plot multiple lines by passing multiple vectors; the default colors will be used
  - `xlabel(name)` – use `name` to label the x axis
  - `ylabel(name)` – use `name` to label the y axis
  - `imshow(mat, origin="upper")` – create image plot of matrix `mat`, with the origin (element `[0][0]`) as the upper left corner (also "lower" left corner option)
  - `axis("off")` – don't display the axes; good for images
  - `colorbar(ticks=range(a,b,s))` – add a colorbar legend; you can also label the legend: `.set_label("label")`
  - `show()` – display the plot figure
- They must all be called with the module name (or a shortcut if you use `import as`).

## ASSIGNMENT 6 WARM-UP

- Let's look at the first part to get a feel for how to use `pyplot`.

# NUMPY

- This library is used for fast scientific computing, including integration with C, C++ and Fortran. It is a core package in the **Scipy** software stack, along with matplotlib and others.
- Instead of using built-in lists, it works directly with arrays.
- An array is a collection of objects, **all of the same data type**.
- We still use indices to access individual elements, and arrays share many of the same features as lists.

## MORE NUMPY ARRAYS

- Arrays can have any number of dimensions, also called the rank of an array. (1D is like a list, 2D is a matrix (nested list), etc.)
- We can access and change the dimensions of an array in numpy.
- We can initialize an array with lists or the `arange` function (array version of `range`).
- See python script example: `npArrays.txt`
- [http://wiki.scipy.org/Tentative\\_NumPy\\_Tutorial](http://wiki.scipy.org/Tentative_NumPy_Tutorial)

## ASSIGNMENT 6

- Let's create some heatplate images with pyplot and numpy.

## WEEK 8: 10/26-10/28

- Object Oriented Programming!
  - using objects, terminology
  - defining our own data types
  - protecting data
  - polymorphism: inheritance
- Quiz 2
- Assignment 7

# OBJECTS & CLASSES

- In Python, all data values are objects.
- Objects are instances of abstract data types defined by classes.
- Classes define the components of each object instance:
  - Attributes – the fields (data) of the object
  - Behaviors – the functions (methods) that can be applied to the objects.
- This is the heart of object oriented programming!

## BUILT-IN TYPES (CLASSES)

- We've used primitive object types (int, float, bool) which have only one data value (field) associated with each object.
- We've used containers which hold multiple values: sequences (str, list, tuple) and sets. (Dictionaries are yet to come.)
- When we use the dot operator to call a function on an object, we're really calling a method defined for that class type.

## TURTLES – OO STYLE

- Instead of having only one turtle, we can create and manipulate several in one window.
- First import the class from the module:
  - `from turtle import Turtle`
- Next create a few turtles and set their colors:

```
t1 = Turtle()
t1.color("green")
t2 = Turtle()
t2.color("blue")
```
- You can then proceed to move them around the window to draw, using all the same functions we already know.
- See `flock.py` for full code

## DESIGNING OUR OWN CLASSES

- Suppose you wanted to run a poker tournament, on-line. What types of objects would you need to represent to handle multiple players, multiple tables, with betting?
- When doing an OO design, we focus first on the nouns in the problem to formulate our classes. Then the verbs become the methods in relevant classes.

## DEFINING A CLASS TYPE

```
class Name:
```

```
    # constructor is used to initialize the
```

```
    # objects we create with this new type:
```

```
    def __init__(self, a, b):
```

```
        self.apart = a
```

```
        self.bpart = b
```

```
        self.cpart = 10
```

```
n1 = Name(4, 'b')
```

```
n2 = Name(13, 'wer')
```

## DISSECTING THE CONSTRUCTOR

- It must be defined as `__init__` with `self` as the first parameter.
- The keyword `self` is used to denote the object to which a method is applied, and appears in various places:
  - As the first parameter of every constructor and instance method.
  - To explicitly refer to the fields of a class type.

## GETTING DATA OUT OF A CLASS

- Typically we define a special method that will convert the objects that are instances of a class into a string:

```
def __str__(self):  
    return self.apart + ' ' + self.bpart
```

- You can include as many fields of a class as desired in its string representation.
- Once you define this, you can say:

```
print 'n1 is ' + str(n1)
```

## MORE INSTANCE METHODS

- We define accessors to get values out of our objects:

```
def getA(self):  
    return self.apart
```

Call: `n1.getA()`

- We can create mutators to change values inside our objects:

```
def setB(self, newb):  
    self.bpart = newb
```

Call: `n2.setB("other")`

## SOME EXAMPLES

- Card Class (Card.py, CardTest.py)
- Time class (Assignment 7 – part 1)

## MORE INSTANCE DATA

- We can also define and initialize containers within our classes:

```
class Course:
    def __init__(self, numb, name):
        self.num = numb
        self.title = name
        self.students = []
```

- Example: CardHand.py

## NAMING CONVENTIONS WITHIN CLASSES

- We use single or double underscores at the start of instance data variable names and class methods to hide them from direct use by clients (other programs using the classes).
- This enables us to prevent direct manipulation of data members that would bypass data validation, and also to indicate that certain methods are "helpers" to be used internally by the class only, not called externally.

## QUIZ 2

- functions, modules, doctest
- lists, nested lists, pyplot, numpy
- recursion

## ASSIGNMENT 7

- Let's define and use our own classes!

## WEEK 9: 11/2 & 11/4

- Operator Overloading
- PyGame for animation and graphics
- Assignment 8
  - get partners!
  - start early

## EQUALITY OF OBJECTS

- There are two built-in ways to compare objects for equality in Python:
  - `valA is valB` – the `is` operator checks for object identity, in other words, whether they are the exact same object in memory. (`valA is not valB` gives the opposite result.)
  - `valA == valB` – the equals operator is defined by a method `__eq__`.
  - `!=` is not necessarily the opposite of `==`; it must be explicitly defined also as method `__ne__`

## COMPARING OBJECTS

- If we define a general purpose special method called `__cmp__(self, other)` for our classes, it will be used when the various comparison operators are used with objects of our class (`<`, `>`, `<=`, `>=`, etc.)
- This method is expected to return an integer:
  - A negative value means `self < other`
  - 0 means `self == other`
  - A positive integer means `self > other`

## MORE OPERATOR OVERLOADING

- Being able to define what code is executed when an operator is used with our objects is a great feature of many object oriented languages called *operator overloading*.
- There are many more operators we can define in Python, but with some subtleties. For more details see section 3.4.1 in:  
<http://docs.python.org/2/reference/datamodel.html>

# INHERITANCE

- We can *extend* an existing class to create a new one for a more specific subclass of objects.
- Examples: table extends furniture, cheeseburger extends sandwich, CardDeck extends CardHand (see these files for details).
- The original class is called the *base* class, and the new one the *derived* class.
- The derived class inherits all the data and methods from the base class, and can add its own data and methods, as well as *override* how methods from the base class act.

# INTERACTIVE GRAPHICS - PYGAME

- install python-pygame (and everything that goes along with it)
- various modules available:
  - display, draw – very versatile & essential
  - sprite, joystick, mixer – more for gameplay
  - image, event, key, many others... – come in handy
- <http://www.pygame.org>

## PYGAME SET-UP

- `import pygame` – imports all modules
- `import pygame.display as PD` – gives an abbreviated name (PD) to that module
- you might want to do that for each module you are using
- `pygame.init()` – must be called before any other pygame code! put it right after your imports, or at the start of main

# PYGAME SURFACES

- Every visual is created with a new Surface.
- The main display window is a special surface we create at the start , optionally specifying the size (width, height) in pixels:

```
winSurf =  
    pygame.display.set_mode((width,height))
```

- Surfaces are essentially matrices of pixels – each with a color. The top left corner is (0,0) and (x,y) means over by x pixels and down by y pixels.
- We can fill our a surface with a color:  

```
winSurf.fill(pygame.Color(255,0,0))
```
- Then we must show it by writing the surface to the screen:

```
pygame.display.flip()
```

## DRAWING SHAPES

- The draw module has built-in functions to draw geometric shapes on a surface, specifying the shape color, position, size, and line thickness (use 0 to fill):

```
pygame.draw.circle(winSurf, ablue,  
    (10,20), 50, 3)
```

- Recall that colors are created with a combination of red, green, blue (rgb) values, each from 0 (none) to 255 (lots):

```
ablue = pygame.Color(0, 0, 180)
```

## TIMING ANIMATIONS

- We can control how quickly our animations update.
- FPS – frames per second – this is the refresh rate for updating or flipping your display
- more frames creates smoother graphics
- sometimes the animation is too processing intensive to go as fast as you would like
- To set it to 60 for example:

```
timer = pygame.time.Clock()  
timer.tick(60)
```

# EVENT-DRIVEN PROGRAMMING

- First we create a loop
- Then we check for events we care about
- We take action for those events
- Usually some event tells us to stop the loop somehow (`pygame.quit()`)
- We update our display (`pygame.display.update()` or `flip()`)

## EVENT TYPES

- pygame.event module contains many constants for different event types
- each Event type has specific read-only data attributes that we can access
- there is an event queue that keeps track of all unprocessed events that have occurred
- we pull events off the queue and perform the actions we've associated with them
- <http://www.pygame.org/docs/ref/event.html>

# EVENT EXAMPLES

```
while True:
    surface.fill(BLACK)
    pygame.display.flip()
    # will return a list of events, could be empty:
    events = pygame.event.get()
    for event in events:
        if event.type == pygame.QUIT:
            pygame.quit()
            sys.exit()
        elif event.type == pygame.MOUSEMOTION:
            if drawing:
                # draws circles where mouse moves:
                pygame.draw.circle(surface, WHITE, event.pos, 4, 0)
        elif event.type == pygame.MOUSEBUTTONDOWN:
            drawing = True
        elif event.type == pygame.MOUSEBUTTONUP:
            drawing = False
        else:
            print event
```

## MORE PYGAME SURFACES

- You can explicitly create a surface of any size:  
`mySurf = pygame.Surface(width, height)`
- You can create surfaces from images:  
`catSurf = pygame.image.load('myCat.png')`
- We can copy one surface (catSurf) onto another (winSurf) by specifying the where the top left corner (of catSurf) will be positioned on the main surface (winSurf):  
`winSurf.blit(catSurf, (posx, posy))`
- You can access many different attributes of surfaces: <http://www.pygame.org/docs/ref/surface.html>

## PYGAME RECTS

- Pygame uses Rect objects to store and manipulate rectangular shape dimensions (pygame.Rect is the class name).
- Initialized with (left, top) – x,y coordinates of the top left corner, and (width, height)
- The position coordinates are used to determine where a Rect is placed relative to a surface.
- They have many attributes we can access and modify, either with new Rects as results, or in place to change the original.
- <http://www.pygame.org/docs/ref/rect.html>

## SURFACES & RECTS

- Surfaces have rectangles (Rect objects) which we can access:

```
catRect = catSurf.get_rect()
```

- We can also blit from one surface to another by using a Rect as the position:

```
winSurf.blit(catSurf, catSurf.get_rect())
```

- The top, left coordinates of the Rect in any Surface is always (0,0).
- A Rect is only a rectangular area specification, whereas a Surface contains colors for the all the pixels in its Rect.

## RECTS & RECTS

- Rects have several useful methods for determining relationships between them and other graphical objects.
  - `rectA.contains(rectB)` – true if rectB inside rectA
  - `rectA.contains(x,y)` – true if point (x,y) is strictly inside rectA (not on edge)
  - `rectA.colliderect(rectB)` – true if they overlap somewhere, false if adjacent or not touching
- There are lots more – look on-line.

## ASSIGNMENT 8

- 2 people per project
- Decide which game to implement
- Get started on part A – game details and design approach!

## WEEK 10: 11/9 & 11/11

- PyGame & inheritance
- Common array operations
- Sorting & searching algorithms
- Algorithm efficiencies
- Assignment 8 (game project) continued

## SURFACES & RECTS

- Surfaces have rectangles (Rect objects) which we can access:

```
catRect = catSurf.get_rect()
```

- We can also blit from one surface to another by using a Rect as the position:

```
winSurf.blit(catSurf, catSurf.get_rect())
```

- The top, left coordinates of the Rect in any Surface is always (0,0).
- A Rect is only a rectangular area specification, whereas a Surface contains colors for the all the pixels in its Rect.

## RECTS & RECTS

- Rects have several useful methods for determining relationships between them and other graphical objects.
  - `rectA.contains(rectB)` – true if rectB inside rectA
  - `rectA.contains(x,y)` – true if point (x,y) is strictly inside rectA (not on edge)
  - `rectA.colliderect(rectB)` – true if they overlap somewhere, false if adjacent or not touching
- There are lots more – look on-line.

# INHERITANCE

- We can *extend* an existing class to create a new one for a more specific subclass of objects.
- Examples: table extends furniture, cheeseburger extends sandwich, CardDeck extends CardHand (see these files for details).
- The original class is called the *base* class, and the new one the *derived* class.
- The derived class inherits all the data and methods from the base class, and can add its own data and methods, as well as *override* how methods from the base class act.
- Example: CardDeck extends CardHand

# PYGAME SPRITES

- <https://www.pygame.org/docs/ref/sprite.html>
- Meant to be used as a base class for objects you want to animate.
- Extend the sprite class using inheritance to define your own customized classes with the core functionality built-in.

# COMMON ARRAY OPERATIONS

- Copy
- Resize
- Insert
- Delete
- Search
- Sort

# ALGORITHM EFFICIENCY

- Measured as functions of the problem size
- Usually do a worst case analysis
- Space
  - how much (extra) memory is used up?
  - recursive methods can be bad in this respect
- Time
  - primary way we compare algorithms
  - how many basic operations (=, arith, compare, etc.)
  - overall tells us how fast or slow is the algorithm
- Big-Oh: upper bounds on efficiency functions

# SEARCHING ARRAYS

- If array is in no particular order:
  - Linear Search
  - go item by item until found, or reach end
- If array is in a particular order (sorted):
  - Binary Search
  - fastest way to win the hi-lo game
  - pick middle value, go left or go right or found

# SEARCHING EFFICIENCIES

- For array problems, size  $N$  elements in array
- Worst case: value is not there
- Linear Search
  - must compare to all  $N$  elements
  - $O(N)$
  - called "linear time", hence "linear search"
- Binary Search (only works if array is sorted)
  - each comparison eliminates  $1/2$  the collection
  - # comparisons = # times can divide  $N$  by 2
  - $O(\log_2 N)$

# SORTING ALGORITHMS

- Insertion sort: consider each element, move to left as far as it needs to go
- Selection sort: find next largest, swap into position, repeat
- Bubble sort: compare adjacent values, swap if out of order, largest values bubble to the end
- Mergesort: split collection in half, mergesort each half, merge them together
- QuickSort: split collection by comparing to a pivot value, recurse (take Data Structures)
- Bucket sort: bins for particular values, sort bins (take Data Structures)

# INSERTION SORT

sorted

unsorted

13            20 5 3 16 4 30 8 3 15 10 6

13 20            5 3 16 4 30 8 3 15 10 6

5 13 20            3 16 4 30 8 3 15 10 6

3 5 13 20            16 4 30 8 3 15 10 6

3 5 13 16 20            4 30 8 3 15 10 6

3 4 5 13 16 20            30 8 3 15 10 6

3 4 5 13 16 20 30            8 3 15 10 6

3 4 5 8 13 16 20 30            3 15 10 6

3 3 4 5 8 13 16 20 30            15 10 6

Etc.

# BUBBLE SORT

unsorted

13 20 5 3 16 4 30 8 3 15 10 6

13 5 3 16 4 20 8 3 15 10 6

5 3 13 4 16 8 3 15 10 6

3 5 4 13 8 3 15 10 6

3 4 5 8 3 13 10 6

Etc.

sorted

30

20 30

16 20 30

15 16 20 30

## SELECTION SORT

13 20 5 3 16 4 30 8 3 15 10 6  
3 20 5 13 16 4 30 8 3 15 10 6  
3 3 5 13 16 4 30 8 20 15 10 6  
3 3 4 13 16 5 30 8 20 15 10 6  
3 3 4 5 16 13 30 8 20 15 10 6

Etc.

# MERGE SORT

split phase

13 20 5 3 16 4

30 8 3 15 10 6

13 20 5      3 16 4

30 8 3      15 10 6

13 20      5 3 16      4

30 8      3      15 10      6

13    20    5    3    16    4

30    8    3    15    10    6

merge phase

13 20      5    3 16      4

8 30      3      10 15      6

5 13 20      3 4 16

3 8 30      6 10 15

3 4 5 13 16 20

3 6 8 10 15 30

3 3 4 5 6 8 10 13 15 16 20 30

## SORTING EFFICIENCIES

- For array problems, size  $N$  elements in array
- Bubble Sort
  - $N-1 + N-2 + \dots + 2 + 1$  ops =  $N(N-1) / 2 = O(N^2)$
- Selection Sort
  - $N-1 + N-2 + \dots + 2 + 1$  ops =  $N(N-1) / 2 = O(N^2)$
- Insertion Sort
  - $1 + 2 + \dots + N-2 + N-1$  ops =  $N(N-1) / 2 = O(N^2)$
- MergeSort
  - $N$  (comparisons/assignments) \* # levels ops
  - # levels = # times can divide  $N$  by 2 =  $\log_2 N$
  - $O(N \log_2 N)$
  - significantly faster than the others

## ASSIGNMENT 8

- Game project work continued

## WEEK 11: 11/16 & 11/18

- Quiz 3
- Dictionaries
- Working with URLs
- Working with Excel files
- Assignment 9

## QUIZ 3

- defining and using our own classes
- pygame basics

## DICTIONARY BASICS

- ✎ This data type enables us to create a mapping of keys to values, stored in a list-like structure:

```
grdPts = {'A':4, 'B':3, 'C':2, 'D':1, 'F':0}
```

- ✎ Think of a dictionary which maps a word (key) to its definitions (value).

- ✎ The keys act as indices into the set of values:

```
grdPts['B'] == 3    # True
```

```
grdPts['B'] = 2.0    # changes the value
```

```
grdPts['C+'] = 2.3    # adds the key & value
```

- ✎ Keys must be of an immutable data type such as strings, numbers, or tuples.

- ✎ We can also get just the set of keys or the set of values in the dictionary as lists:

```
grdPts.keys()    => ['A', 'B', 'C', 'D', 'F']
```

```
grdPts.values()   => [4, 3, 2, 1, 0]
```

## WORKING WITH DICTIONARIES

- ✎ Iterate through the keys to do something with each value in the dictionary:

```
for key in grdPts.keys():  
    print key + ' grade points: ' + str(grdPts[key])
```

- ✎ Search for a key or a value in the dictionary:

```
if key in grdPts: # assumes we mean grdPts.keys()  
if value in grdPts.values():
```

- ✎ Create an empty dictionary:

```
dict = { }
```

- ✎ Delete a key (& its value) from a dictionary:

```
del grdPts[key]
```

- ✎ Add a new key:value pair to the dictionary:

```
if key not in dict:  
    dict[key] = value
```

- ✎ Note that dictionaries are unordered!

## GETTING WEB DATA

- ✎ A URL is a Uniform Resource Locator and we use the urllib module in Python to access webpages.
- ✎ Check that urllib is installed on your VirtualBox and remember to import urllib in files that use it.
- ✎ We use the urlopen function to create a file object from a URL:

```
file = urllib.urlopen('http://www.cs.jhu.edu/  
~joanne/cs112/lects/courses.txt')
```
- ✎ If you open an html file, it will read all the source html commands along with the plain text.
- ✎ We can then use standard file functions to read the page source.

## MORE FILE READING

- ✎ Once you open a file (whether a plain text one in your file system or from a URL), there are several ways to read the data.
- ✎ Do something with each line: `for line in file`
- ✎ Get the entire file as one long string with linebreaks:  
`page = file.read()`
- ✎ Get the next line of the file:  
`line = file.readline()`
- ✎ Get a list of the lines in the file:  
`linelist = file.readlines()`
- ✎ The read functions also have versions where you can use an integer parameter specifying how much to read (characters or lines)

## WORKING WITH EXCEL FILES

- ✎ Python has libraries which let us work directly with data in Excel files: [www.python-excel.org](http://www.python-excel.org)
- ✎ Need to install python-xlrd and python-xlwt through the Synaptic Package Manager
- ✎ xlrd is for reading data from Excel files
- ✎ xlwt is for writing data to Excel files
- ✎ To open an Excel workbook called *filename*:  

```
wb = xlrd.open_workbook(filename)
```
- ✎ To create an Excel workbook and add a sheet:  

```
wb = xlwt.Workbook()  
ws = wb.add_sheet('Data')  
wb.save(filename)
```

## READING FROM EXCEL FILES

✎ Processing all the sheets:

```
for s in wb.sheets():  
    print 'Sheet:' + s.name
```

✎ Processing all rows in a particular sheet:

```
s = wb.sheet_by_index(0)  
for r in range(s.nrows):  
    for c in range(s.ncols):  
        print s.cell_value(r,c)
```

## WRITING TO EXCEL FILES

✎ Create and save a workbook:

```
wb = xlwt.Workbook()  
wb.save(filename)
```

✎ Create a sheet:

```
ws = wb.add_sheet('Data')
```

✎ Put data (val) in the cell at (row,col):

```
ws.write(row,col,val)
```

✎ Get and use a row:

```
row = ws.row(1)  
row.write(col, val)
```

✎ Lots of other options and features exist!

## EXAMPLE – COURSE PROCESSING

- url for text file:  
<http://www.cs.jhu.edu/~joanne/cs112/lects/courses.txt>
- python program to process:  
<http://www.cs.jhu.edu/~joanne/cs112/lects/courses.py>
  - reads from URL
  - writes to excel file
- excel file: <http://www.cs.jhu.edu/~joanne/cs112/lects/courses.xls>

# ASSIGNMENT 9 – COURSE "DATABASE"

## ○ Classes:

- CourseID
- Course (has CourseID)
- CourseListing (is/extends Course)
- SemesterListing (has CourseListing(s))
- CoursesMain (uses all)

## ○ To Do:

- finish SemesterListing – work with dictionary of Courses and inheritance
- finish CoursesMain – work with opening URLs for file input, loading and saving to Excel files

## WEEK 12: 11/30 & 12/2

- Testing (review)
- Exception handling
- Quiz4
- Game Demos!

# TESTING, REVISITED

- White-box Testing: each possible path in a program (all possible decision cases) should be tested.
- Black-box Testing: test problem requirements, ignoring code
- Boundary cases (the = part of  $\leq$  or  $\geq$ ) should be tested.
- Valid values should be tested.
- Invalid values should be tested.

## TESTING, CONTINUED

- Regression Testing: when rewriting and updating code, be sure to re-test cases that worked before the "upgrade".
- Unit Testing: individually test each function using doctest or other means – incorporate white-box testing
- Testing Data
  - make up literal values
  - random values – helps with black-box testing
  - loops to generate data

# ERROR HANDLING

- dealing with run-time errors: invalid data, file problems, invalid operations (eg – convert 'three' to an int)
- test, display error message(s)
- test, bail out of the program:
  - `sys.exit('some error message')`
- test, use exception handling
  - <http://docs.python.org/2/tutorial/errors.html>

# EXCEPTION OBJECTS

- exceptions are objects that represent error conditions that have occurred
- there is an inheritance hierarchy of exception classes that are built into Python for common issues: <https://docs.python.org/2/library/exceptions.html#exception-hierarchy>
- we can create our own custom exception classes too, but often that is not necessary

# BUILT-IN PYTHON EXCEPTION CLASSES

Specific common errors:

- `ZeroDivisionError` : `val / 0`
- `NameError` : `print 3*num` # num not initialized
- `TypeError` : `'name' + 23` # str + int not good
- `ValueError` : `int("not an int")` # casting error
- `IOError` : if you try to open a file that isn't there

More general exception classes:

- `RuntimeException`
- `Exception`

## EXCEPTION HANDLING (IN BRIEF)

- create a `try` block – inside there:
  - call code that generates an exception
  - explicitly `raise` an exception object
- catch an exception object to handle it gracefully with an `except` clause following the `try` block
  - you can have multiple different except clauses to catch different exception types, or one that catches several listed in a tuple
- use `finally` clause to execute code no matter what (regardless of whether an exception has been raised)
- if an operation "`raise`"s an exception that is not caught, eventually the program will stop and list the open function calls at that point in the program execution, called a Traceback

## EXAMPLES

- `except.py` (linked from `schedule`) has several small examples that show the flow of control between methods with exception handling
- `CourseID.py` (linked from `schedule`) has exception handling for course number part verification.
- Let's add it to `courses2.py` for I/O handling.

## QUIZ 4 (12/2)

- Searching, sorting, efficiency, dictionaries, testing, exceptions (Zyante chapters 7.9-7.15, 15)

# GAME DEMOS!

- Show off your hard work!