CS50 Section 8
Somewhere in Between

Annaleah Ernst, TF

Agenda

- Python
  - misc.
  - variables
  - conditionals
  - loops
  - arrays and lists
  - tuples
  - dictionaries
  - functions
  - objects

Python 3.0

- A higher order programing language
- A lot closer to pseudocode than C
- There are a lot of tutorials everywhere
  - AND they're Monty Python themed
    - That's how Python got its name
  - Careful of what version resources are using
  - The course uses version 3; 2.7 is the most common, but there are some differences
  - If things you're getting from online aren't working, probably the wrong py version
- Uses *whitespace* to separate code
- No more semicolons, no more curly braces!

Misc

- Analogous to C's #include in C is import
  - `import cs50` to import the module
  - `from cs50 import get_int` to import a specific function
- There is no ++ operator
  - you must use `var += 1` instead
- No statements end in semicolons!
- Conditional blocks, functions, and loops are not denoted with curly braces
- // does integer division
- / does floating point division (regardless of type)

Variables

- Declared by assignment
- No *explicit* types
- There are implicit types
  - number
  - string
  - list
  - char
  - tuple
  - dictionary
- See 0-string and 1-temperature in the supplemental code!

- C
  - `int coursenum = 50;`
  - `string coursename = "CS50";`
- Python
  - `coursenum = 50`
  - `coursename = "CS50"`

Printing

- Print statements are terminated with a newline unless otherwise specified using the `end` keyword argument
- Print statements are a full blown function in Python 3, and thus arguments must be surrounded in parenthesis
  - Careful of online tutorials using Python 2.7; the print function was different

Comments

- Similar rules to C, but now, we only have the # symbol instead of // and /**/
- Note that `"""this"""` is a docstring, not a comment
  - We'll talk about these when we get to functions

Conditionals

- Conditionals just got a whole lot more English!
- We use the keywords `and`, `or`, and `not` instead of their symbolic representation
- `else if` from C is now `elif` in Python
- No more curly braces! But colons just got a lot more important
- Code subject to a conditional MUST be indented or it WILL NOT WORK
  - We end the conditional block by returning to the previous level of indentation
- Take a look at 2-conditions and 3-logical in the supplemental code

Loops

- For loops
  - really, these behave like foreach loops
  - ```
    for thing in things:
        print(thing)
    ```
  - These are really versatile and flexible compared to C counterpart
- While loops
  - These are just like their C counterparts
- Note that there is no do-while equivalent
- See 4-quack and 5-argv in the supplemental code
    ```

Lists (formerly Arrays)

- In Python, we have lists instead of arrays
  - More or less equivalent, but lists are more powerful
- Create using square bracket notation
  - `mylist = [1,2,3,4]`
- Unlike C, contents is not required to be the same type!
  - Generally though, for design's sake, it should be!
- There are built in methods that will help you mutate the list (ie, change it in place)
- We'll take a look at 6-strings and 7-capitalize

Lists – In Place Methods

- `mylist.append(value)`
  - add another value to the end of the list
- `mylist.insert(index, value)`
  - add value at a location
- `mylist.extend([elt1, elt2, etc])`
  - add a list ([value, value, value]) to the end of the current list
- `mylist.sort()`
  - sorts list in place

Lists – Other methods

- `sorted(list)`
  - returns a sorted copy of the list
- `mylist + [value]`
  - returns a copy of the list with value appended to it
- `[value] * number`
  - create a list containing value number times
- `len(list)`
  - returns the number of element in list

Strings

- Strings are not mutable; you cannot make changes to elements of a string
- However, if you iterate over their elements, you get characters
- You can cast strings to lists
  - `list("hi")`
  - `['h', 'i']`
- If you cast a list to a string, you get something very literal
  - `str(['h', 'i'])`
  - `"['h', 'i']"`
- We can concatenate strings using the + operator

Strings (extra, just for fun)

- You can use join to combine all the elements in a pretty string
  - `"<separator character>".join(mylist)`
  - returns the elements in you list as a string separated by the separator
- As with lists, I can create a string with a repeated element by via
  - somestring * num_repetitions
  - eg, `'a' * 5` returns `aaaaa`

Characters

- In python, the chr type has several built ins you might find useful
- mychr.upper()
  - returns upper case version of character
  - use mychr.lower() to get the lowercase version
- ord(mychr)
  - returns the ASCII value of the letter
- chr(myint)
  - returns the character represented by the ASCII value

Tuples

- Data type for ordered, immutable data
  - You can access members of the tuple, but you can’t change those members directly
- Declared using parenthesis
- These can also be iterated over
  - ie, you could put them in a Python for loop

Dictionaries

- Effectively the equivalent of a hash table in C
  - or, an array whereby you index in using keywords
- Consist of key-value pairs
  - key is an int, string, or chr
  - value is anything (including other dicts!)
- Assign a set of key-value pairs in curly braces
  - {‘key’ : ‘value’, ‘nextkey’ : ‘nextval’}
- You cannot have duplicate keys
- check out 9-speller in the supplemental code

Functions

- No return types! We just define using the ‘def’ keyword
- Like C, they have names and parameter lists
  - `def myfunction(arg1, arg2):`
    - `# code`
- files are *interpreted* not compiled
  - They are read top to bottom left to right
  - We can’t prototype functions the way we could in C
- Code does not have to bound up in a main() function, though we will often emulate that syntax with some extra syntax
  - This is the only way to simulate prototyping
- Functions can return multiple values and any type of data
- Check out 9-positive and 10-cough in the supplemental code

Functions cont

- To add a main function, we define all of our helper functions
- Then define some function that you want to behave like main
  - nothing special yet!
- Then at the very bottom of the program we write
- `if __name__ == "__main__":`
  - `# call the function that we want to be our main-like function!`
- Don't worry about the if line; we'll talk about it later
- In stead of the block comments above the function like in C, we use doc strings right below the function declaration
  - `def myfunction(arg1):`
    - `"""Info about the function goes here."""`
    - `# the code goes down here`

Scope

- Things don’t scope quite the way they used to…
- Python is a lot less picky about what you’re doing
  - DON’T LET THAT MAKE YOU COMPLACENT
- There’s a lot more opportunity to introduce unexpected bugs
- Careful with global variables
  - we should declare these explicitly within functions
- See 12-global in the supplements

Objects

- These are like C structs on steroids
  - Not only do they have fields, but al also methods!
- Created using the class keyword
- Define methods and properties inside of the class
- Class must contain at least one special method: called __init__
- The first parameter of every method in a class definition must be self
  - …but not when you call it!
  - To call methods on an object, we do myclass.method()
    - the self is implicit in the dot syntax!
- See 11-objects in the supplemental code