CISC101 Reminders & Notes

- Discussion: lab and tutorial sections
- Labs and tutorials start tomorrow
  - Meet your TA
  - Work on the lab for Week 2
  - TAs can answer questions
- Assignment 1 has been posted

Slides courtesy of Dr. Alan McLeod

Today

- How can we tell the CPU what to do?
- Where Python comes from
- Language fundamentals

Slides courtesy of Dr. Alan McLeod

Commanding the Processor

- Suppose we want the processor to carry out an operation
  - `X = A * B + C`
- Assume we have used some other operations to put numbers in three memory locations in RAM
  - `A` is at address `1024`
  - `B` is at address `1025`
  - `C` is at address `1026`
- We want the result to go into memory location `1027` for `X`

Slides courtesy of Dr. Alan McLeod

X = A * B + C

- Remember the “von Neumann Cycle”?
- The operations in the ALU (or “Arithmetic Logic Unit”) part of the CPU would be ...
  1. Fetch the contents of location `1024` (`A`)
     - Put the value into a register
  2. Fetch the contents of location `1025` (`B`)
     - Multiply it with the value (`A`) in the register
     - Store the result (`A * B`) in the same register
  3. Fetch the contents of location `1026` (`C`)
     - Add this value to the contents of the register (`A * B + C`)
  4. Move the contents of the register to location `1027` (`X`)

Slides courtesy of Dr. Alan McLeod

X = A * B + C, Cont.

Naturally, these instructions have to be communicated to the CPU in binary ...

1. 00010000 00000000000001000000000
2. 00100100 00000000000001000000001
3. 00100011 00000000000001000000010
4. 00010001 00000000000001000000011

byte 1 | bytes 2, 3 and 4

opcode | **operand**

Slides courtesy of Dr. Alan McLeod

X = A * B + C, Cont

- What are these instructions in base 10?

1. 16 1024
2. 36 1025
3. 35 1026
4. 17 1027

- The operands are the memory locations
- The opcodes are 16 for *load*, 36 for *multiply*, 35 for *add* and 17 for *store*
- The default register is used
  - Does not need to be specified

Slides courtesy of Dr. Alan McLeod

Machine Language

- The 4-byte binary commands are examples of machine language
- Normally these commands would be viewed in base 16, or hexadecimal

1. 0x10000400
2. 0x24000401
3. 0x23000402
4. 0x11000403

Slides courtesy of Dr. Alan McLeod

X = A * B + C, Cont

- People have problems remembering binary codes and even decimal codes for operations
- A “shorthand” language called Assembly Language was introduced
  - Works at a level above machine language

Slides courtesy of Dr. Alan McLeod

X = A * B + C, Cont.

- (Pseudo) Assembly language instructions:

```
1. LOAD A, ACC
2. MULT B, ACC
3. ADD C, ACC
4. STOR ACC, X
```

- Each assembly language keyword is translated into its corresponding machine language code
  - Done by an "interpreter" program called an Assembler

Slides courtesy of Dr. Alan McLeod

X = A * B + C, Cont.

- A "high-level" language goes one step above assembly language
- In C, C++, C# or Java you would write

```
X = A * B + C;
```

- Python is the same, except no semi-colon
- Each high-level line of code gets translated into many lines of assembly code
  - Each line of assembly code must then be translated into binary machine language
- Much easier to write, yes?

Slides courtesy of Dr. Alan McLeod

Aside - A Program in C and Assembly

Here is a (very) short C program with our calculation:

```
int main() {

  int a = 1, b = 2, c = 3, x;

  x = a * b + c;

  return 0;

}
```


Slides courtesy of Dr. Alan McLeod

Aside - Cont.

Here is the (real) Assembly language portion for just the **`x = a * b + c;`** line:

```
mov    0xfffffffc(%ebp),%eax
imul   0xfffffff8(%ebp),%eax
mov    0xfffffff4(%ebp),%edx
add    %eax,%edx
mov    %edx,0xfffffff0(%ebp)
```


Slides courtesy of Dr. Alan McLeod

Computer Languages - Cont.

- Each assembly language command is translated into several machine language commands
- The next generation of computer languages went up one more level
  - Got closer to something readable
  - *e.g.*, Fortran, Cobol and Lisp
- These languages led to an explosion of over 200 languages being developed in the 60s and 70s
  - *e.g.*, Basic, Pascal, C, Ada and Smalltalk
- Python is a relative newcomer, arriving on the scene in the early 90s

Slides courtesy of Dr. Alan McLeod

History of Python

- The language was created by Guido van Rossum at Stichting Mathematisch Centrum in the Netherlands in the early 90s
- He is still very involved with the language
  - Retains the title BDFL
    - Benevolent Dictator for Life
- Python is named after “Monty Python”, not the snake!

Slides courtesy of Dr. Alan McLeod

History of Python - Cont.

- He wanted to make the language
  - easy and intuitive
    - … but just as powerful as major competitors
  - open source
    - anyone can contribute to its development
  - use code that is as understandable as plain English
  - to be suitable for everyday tasks
- First released in 1994, the language was inspired by Modula-3, Lisp, SETL, Haskell, Icon and Java
  - A compilation of the “Best-Of’s” from many other languages!

Slides courtesy of Dr. Alan McLeod

Features of Python

- High Level
  - Most notable are the built-in data structures
- Object Oriented
  - OOP helps you to build code in a modular way
  - Python allows you to write code without knowing anything about OOP!
- Scalable
  - Packaging of code allows even very large programming projects to be manageable
- Extensible
  - You can easily use external code modules written in Python, C, C++, C#, Java or Visual Basic

Slides courtesy of Dr. Alan McLeod

Features of Python - Cont.

- Portable
  - Runs on any platform/OS combination that can run C
- Easy to Learn (**!**)
  - Relatively few keywords, simple language structure and clear syntax
  - OOP can be avoided while learning
- Easy to Read
  - Much less punctuation than other languages
  - Forces you to use good indentation
- Easy to Maintain
  - Results from the two above features!

Slides courtesy of Dr. Alan McLeod

Features of Python - Cont.

- Robust
  - Exception handlers and safe, sane and informative crashes
- Good for Rapid Prototyping
  - Often used with other languages to create prototypes and provide a testing platform
- Built-In Memory Management
  - Like Java; avoids a major problem in C/C++
- Interpreted
  - Not compiled; each command is executed as it is read from the program
  - Speed is increased using byte-compiled files (like Java)

Slides courtesy of Dr. Alan McLeod

Compiled vs. Interpreted Languages

- *Compilation* means that the machine code translation of a program must be created completely before the program is run
- This machine code is usually saved in a file
  - *e.g.*, an executable (*.exe) or some kind of compiled library of code (*.dll)
- Advantage: these programs can run very quickly
  - Little or no translation is required
- Disadvantage: the editing/testing/debugging loop takes longer
- Languages like C and C++ are compiled

Slides courtesy of Dr. Alan McLeod

Compiled vs. Interpreted Languages - Cont.

- When a program in an *interpreted* language is run, the machine code is generated and executed one line at a time
- No machine code is saved as a file
- Advantage: ease of development and testing
  - Better productivity
- Disadvantage: speed
  - Interpreted code can be up to 10 times slower than compiled
  - Longer *execution times*

Slides courtesy of Dr. Alan McLeod

Compiled vs. Interpreted Languages - Cont.

- A modern trend is to have interpreted languages create a file that is partially compiled
  - A *byte code* file
- This speeds up execution without giving up any of the advantages of using an interpreter
- Python can, and Java does, work this way
- C, C++ and Java programs often run faster than Python programs
  - Many computing-intensive Python operations are "farmed" out to libraries written in C

Slides courtesy of Dr. Alan McLeod

Python!

- When you see the Python prompt

```
>>>
```

  you know you are speaking directly to the interpreter
- If you want to store your commands in a file that can be edited without retyping everything, create a script or program
- The program can be easily fed to the interpreter
  - Executes it one line at a time

Slides courtesy of Dr. Alan McLeod

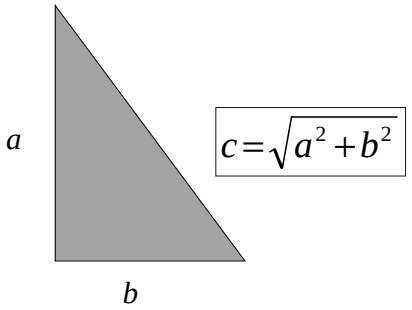
Python Demos

- First one: Ritual!
  - "Hello World" at the command prompt
  - "Hello World" in a script (or "program")
  - "Hello World" in a function in a program
  - Jazz it up a bit:
    - Add a comment or two
    - Get some input from the user
- Second one: triangle calculation

Slides courtesy of Dr. Alan McLeod

Triangle Calculator Demo

- A program that obtains two numbers from the user and then returns the side length of the third side of a right angle triangle



Slides courtesy of Dr. Alan McLeod

Built-In Functions

- Also known as *BIFs*
- Named pieces of code provided by Python
  - Accomplish fundamental and common tasks
  - Used or *invoked* by "calling their names"
- What do some of the BIFs do?
  - Write output to the screen (*e.g.*, `print()`)
  - Convert data from one type to another
  - … and more
- Often provide data for functions inside `()`
  - These *parameters* are separated by commas

Slides courtesy of Dr. Alan McLeod

Literal Values and Data Types

- A *literal value* is a specific data value
  - The value of the data is "literally" itself
  - *e.g.*, 42, 3.1416, "Monty Python", True, …
- Literals must be one of several *data types*
  - `int`
  - `float`
  - `str`
  - `bool` (only possible values are `True` or `False`)
  - … and some others we'll see later

Slides courtesy of Dr. Alan McLeod

Numeric Types

- The **`int`** type is an integer (no decimal or exponent) and there is no limit to its size
- The **`float`** type is characterized by a decimal place and/or an exponent
  - It is limited to about 17 digits
- (We won't use the **`complex`** type much!)

Slides courtesy of Dr. Alan McLeod

Other Bases

- Normally we view numbers in base 10, or in a *radix* of 10
  - That's the default in Python
- How can you view numbers in base 2, 8 or 16?
- Use prefixes `0b`, `0o` or `0x` on literals
- Use the BIFs `bin()`, `oct()` and `hex()`

Slides courtesy of Dr. Alan McLeod

Sequence Types

- The **`str`** type represents a sequence of characters enclosed in …
  - single quotes
  - double quotes
  - three single quotes
  - three double quotes
- We'll look at other sequence types (*e.g.*, `list`, `tuple`) later in the class

Slides courtesy of Dr. Alan McLeod

Variables

- What is a variable anyways?
- In Python, variables are created by an assignment statement (or in function parameter lists)
- A variable takes the type of the value being assigned to it when the program runs
- A variable's value can change at any time
- Can a variable change its *type* at any time in Python? Yes!

Slides courtesy of Dr. Alan McLeod

Assigning/Creating a Variable

- In code:

```
myVal = 20
```

- Now **`myVal`** refers to some location in RAM that stores the **`int`** type value **`20`**
- We don't have to worry about what the actual memory address is!

Slides courtesy of Dr. Alan McLeod

Variable Naming Syntax Rules

- Variable names are case sensitive
- You can't use a Python keyword for a variable name
- No spaces!
- Start with a letter (use lower case, by convention)
  - Can also use underscore _
- The rest of the name can contain numbers, letters or the underscore
  - No spaces ! (*oops I said that already!*)

Slides courtesy of Dr. Alan McLeod

Variable Naming Style Rules

- Use descriptive names
- Capitalize successive words in a name
  - Use *camelCase*
- No limit to the length of a variable name …
  - … but don't write an essay!!
- Don't use single letter variable names
  - Exception: a loop counter that has no intrinsic meaning
  - Then you can use `i`, `j` or `k`

Slides courtesy of Dr. Alan McLeod

Keywords

- Words used by Python for specific purposes
  - Considered part of the Python language
  - *e.g.*, `if`, `while`, `import`, …
- Used to construct programs that can …
  - test conditions
  - perform tasks repetitively
  - use other Python functions
  - … and more!
- Can not use keywords as variable names

Slides courtesy of Dr. Alan McLeod

Arithmetic Operators

- As listed in Lab 1:
  + addition
  - subtraction (and unary negation)
  * multiplication
  / division
  // "floor" division
  % modulo or "remainder"
  ** exponentiation or "to the power of"
- The first four make sense, how do the last three work?

Slides courtesy of Dr. Alan McLeod

/ or Division

- For example, what is the value of `1 / 3` ?

```
0.3333333333333333
```

- In previous versions of Python the result of `int` divided by `int` would be an `int`
  - Not any more!
  - Now the result is a `float`.

Slides courtesy of Dr. Alan McLeod

// or Integer or "Floor" Division

- What is the value of `1 // 3`?

`0`

- The result is always truncated, not rounded
  - So `99 // 100` is also zero
- You still get the truncated value, even if one or the other numbers are `floats`
  - So `1.0 // 3.0` is still `0.0`

Slides courtesy of Dr. Alan McLeod

% or "Modulo"

- Always gives the remainder after division
- For example `20 % 3` is 2
  - `int` values yield an `int` result
- Also works with `floats`
  - Result is a `float`

Slides courtesy of Dr. Alan McLeod

** or "To the Power Of"

- Example: `5 ** 2` is 25
- If both numbers are `ints`, the result is an `int`
  - You can generate some pretty large `int` values this way!
  - Example: `5 ** 100` is **`78886090522101180541172855652827862296732064351090230047702789306640625`**
- If at least one number is a `float`, the answer is a `float`
  - So `5.0 ** 100` is **`7.888609052210118e+69`**

Slides courtesy of Dr. Alan McLeod

Mixed Numeric Types

- If you have an expression like

`aVal = 4 + 5 – 2.0`

the result will always be a **`float`**
  - **`7.0`** in this case.
- So, if you get any **`float`** values in an expression that has integers the result will always be a **`float`**
  - Example: **`4 + 10 / 5`** gives **`6.0`**

Slides courtesy of Dr. Alan McLeod

Strings and Numbers

- Multiplication

  `5 * 'abc'` is `'abcabcabcabcabc'`

- The other numeric operators will not work with strings

Slides courtesy of Dr. Alan McLeod

Addition and Strings

- The **+** sign not only adds numbers but can also *concatenate* strings (and collections)
- Shall we try it out?
- You can concatenate a number to a string, but you have to convert it to a string first using the `str()` BIF

Slides courtesy of Dr. Alan McLeod

Precedence Rules

- Suppose I have an expression like:

```
a = 5 * b + 27 / c
```

- How do we know the order of operations?

****** first, then **-** (as unary negation), then *****, **/**, **//** and **%**, then **+** and **-**, and then **=**

- These rules are built-in to the interpreter
  - **=** is always last (why?)

Slides courtesy of Dr. Alan McLeod

Precedence Rules - Cont.

- Rules again:

******
- (unary)
*** / // %**
+ -
=

All operators on the same line have equal precedence.

Slides courtesy of Dr. Alan McLeod

Precedence Rules - Cont.

- How can you take over control of the order of operations? Use the round brackets!
- Example: `(4 + 5) // 3` is 3
  while `4 + 5 // 3` is 5
- What if you have a series of operators that have equal precedence and no brackets to control things?
  - The expression is evaluated from left to right

Slides courtesy of Dr. Alan McLeod

Indentation

- Press the <tab> key to get an indent in your code, if you need one
- Use the <Backspace> key to get out of an indentation
- Don't type spaces for indents!
- Indents are very important in Python, they are not just "whitespace"!
- IDLE starts indentation automatically, especially after you type a **`:`**

Slides courtesy of Dr. Alan McLeod

Using the `print()` BIF

- Writes the given values on the screen
- You can supply any number of parameters to the function by supplying a comma-separated list of parameters inside the brackets
  - May also include no parameters at all
- Parameters can be variables or literal values
  - Or values supplied by some other function
- Comma-separated values are printed together separated by a space, by default
- What are two ways to avoid this behaviour?

Slides courtesy of Dr. Alan McLeod

Escape Characters

- Can be used to control how strings are displayed when using the **`print()`** function
- See table 2-7 in the textbook
  - `\n` - linefeed
  - `\t` – tab character
  - `\'` – single quote
  - `\"` – double quote
  - `\\` - backslash
- You can also use triple quotes to create a multi-line string without using escape characters

Slides courtesy of Dr. Alan McLeod

Some Punctuation …

- Comments start with the pound sign #
- Long lines can be continued using \
- Put a lower case r in front of a string literal to get a “raw string”
  - Escape characters will not format the string in this case

Slides courtesy of Dr. Alan McLeod