CISC101 Reminders & Notes

- Assignment 2 sample solutions are posted
- Test 2 takes place this week <u>in tutorial</u>

Slides courtesy of Dr. Alan McLeod

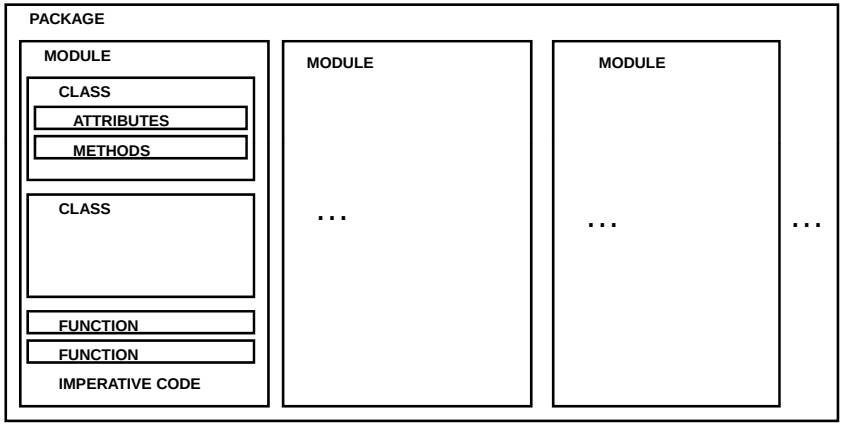
Writing Great Programs

- Two ways to make the best programs
- Modular Programming
  - Using, defining and designing functions
    - Review some of what we learned earlier
- Style and Documentation
  - We've already discussed this …
  - … but there's still more to do

Slides courtesy of Dr. Alan McLeod

Modular Programming

- There are many layers to how code is grouped



Slides courtesy of Dr. Alan McLeod

Operational Code

- There are three places where you can put code that does something
  - In a function
  - In a class
  - Outside of functions and classes
    - Imperative code

Slides courtesy of Dr. Alan McLeod

Imperative Code

- Imperative code is not inside a function or class
  - Written starting in the leftmost column
- We've written imperative code before
  - The first two programs for Assignment 1 were imperative
- We still use imperative statements occasionally
  - **def** *aFunction*(…) :
  - Calling **main()**
  - *etc.*
- The next slide has an example of an imperative program

Slides courtesy of Dr. Alan McLeod

Imperative Program or "Script"

```
# This is the most simple kind of program
# you can write!

for i in range(10) :
    print(i, end=', ')
print("\nAll done!")
```

This script displays the following output:

```
0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
All done!
```


Slides courtesy of Dr. Alan McLeod

Code in Functions

```
# This is the approach necessary for
# Assignment 2 (and from now on)

def printNumbers(stop) :
    for i in range(stop) :
        print(i, end=', ')

def main() :
    printNumbers(10)
    print("\nAll done!")

main()
```


Slides courtesy of Dr. Alan McLeod

Code in Classes

```
# A very simple class with one method

class myClass(object) :

    def aMethod(stop) :
        for i in range(stop) :
            print(i, end=', ')
        print("\nAll done!")

def main() :
    myClass.aMethod(10)

main()
```


Slides courtesy of Dr. Alan McLeod

Modules and Packages

- A module can contain any or all of the following
  - Imperative code
  - Functions
  - Classes
- We've actually been building modules
  - *We just didn't know it*!
- A package is a collection of modules
- In CISC101 we will *not* build classes or packages

Slides courtesy of Dr. Alan McLeod

Modules and Functions

- Goal: break up code into separate functions
  - These are invoked or "called" from **`main()`**
- Let's review what we already know about functions
- Python already comes with a set of built-in functions or BIFs
  - What are they and which ones have we been using?
    - For more information, consult the Python help docs

Slides courtesy of Dr. Alan McLeod

66 BIFs

abs()	delattr()	globals()	list()	print()	sum()
all()	dict()	hasattr()	locals()	property()	super()
any()	dir()	hash()	map()	range()	tuple()
ascii()	divmode()	help()	max()	repr()	type()
bin()	enumerate()	hex()	memoryview()	reversed()	vars()
bool()	eval()	id()	min()	round()	zip()
bytearray()	exec()	input()	next()	set()	
bytes()	filter()	int()	object()	setattr()	
chr()	float()	isinstance()	oct()	slice()	
classmethod()	format()	issubclass()	open()	sorted()	
compile()	frozenset()	iter()	ord()	staticmethod()	
complex()	getattr()	len()	pow()	str()	

Slides courtesy of Dr. Alan McLeod

66 BIFs – The Ones We Use Often

abs()	delattr()	globals()	list()	**print()**	sum()
all()	dict()	hasattr()	locals()	property()	super()
any()	dir()	hash()	map()	**range()**	tuple()
ascii()	divmode()	help()	max()	repr()	type()
bin()	enumerate()	hex()	memoryview()	reversed()	vars()
bool()	eval()	id()	min()	round()	zip()
bytearray()	exec()	**input()**	next()	set()	
bytes()	filter()	**int()**	object()	setattr()	
chr()	**float()**	isinstance()	oct()	slice()	
classmethod()	format()	issubclass()	open()	sorted()	
compile()	frozenset()	iter()	ord()	staticmethod()	
complex()	getattr()	**len()**	pow()	**str()**	

Slides courtesy of Dr. Alan McLeod

All The BIFs We Have Used

abs()	delattr()	globals()	list()	**print()**	sum()
all()	dict()	hasattr()	locals()	property()	super()
any()	dir()	hash()	map()	**range()**	tuple()
ascii()	divmode()	help()	max()	repr()	type()
bin()	enumerate()	hex()	memoryview()	reversed()	vars()
bool()	eval()	id()	min()	round()	zip()
bytearray()	exec()	**input()**	next()	set()	
bytes()	filter()	**int()**	object()	setattr()	
chr()	**float()**	isinstance()	oct()	slice()	
classmethod()	format()	issubclass()	open()	sorted()	
compile()	frozenset()	iter()	ord()	staticmethod()	
complex()	getattr()	**len()**	pow()	**str()**	

Slides courtesy of Dr. Alan McLeod

A Few More We Are Going to Use…

abs()	delattr()	globals()	list()	**print()**	sum()
all()	dict()	hasattr()	locals()	property()	super()
any()	dir()	hash()	map()	**range()**	tuple()
ascii()	divmode()	help()	max()	repr()	type()
bin()	enumerate()	hex()	memoryview()	reversed()	vars()
bool()	eval()	id()	min()	round()	zip()
bytearray()	exec()	**input()**	next()	set()	
bytes()	filter()	**int()**	object()	setattr()	
chr()	**float()**	isinstance()	oct()	slice()	
classmethod()	format()	issubclass()	open()	sorted()	
compile()	frozenset()	iter()	ord()	staticmethod()	
complex()	getattr()	**len()**	pow()	**str()**	

Slides courtesy of Dr. Alan McLeod

Functions and Methods

- A method belongs to, or is a *member* of a class
  - A method is defined within a class
  - A method must be invoked by naming the class (or object) that owns the method
  - *e.g.*, `aString`**`.format(…)`** is a string method
- A function belongs to a module
  - A function is not defined in a class
  - A function is invoked directly
  - *e.g.*, any function you have defined thus far

Slides courtesy of Dr. Alan McLeod

Module Functions

- Just in case we don't have enough BIFs, you can always get more from other modules.
- We have used the `math` and the `random` modules to obtain other functions

`math.sqrt(…)`

`random.randInt(…)`

Slides courtesy of Dr. Alan McLeod

Module Functions - Cont.

- You have to tell the interpreter when you wish to use a function in a module
  - You need to *import* the module

```
import math
```

- What if you don't want to include the module name every time you use the function?
  - It would be nice to just call **`sqrt()`** rather than **`math.sqrt(…)`**
  - Solution: use a different kind of import statement

```
from math import *
```


Slides courtesy of Dr. Alan McLeod

Module Functions - Cont.

- Using **`from math import *`** means you can invoke any of the `math` functions directly
  - **`sqrt(2)`** instead of **`math.sqrt(2)`**
- What if you just want one function from a module?
  - Such as the **`sqrt()`** function

```
from math import sqrt
```


Slides courtesy of Dr. Alan McLeod

Writing Functions

- Function header syntax:

```
def function_name(parameter_list) :
```

- Use the normal naming rules for *`function_name`*
- *`parameter_list`* provides a mechanism for getting values into your function
  - But it's optional
- The **`return`** keyword can be used to send a value out of a function
  - But it's optional

Slides courtesy of Dr. Alan McLeod

Parameters and Arguments

- Invoke functions with zero or more *arguments*
  - Values for the function's parameters
    - "Parameter" and "argument" are often used interchangeably
- Arguments are separated by commas and can be
  - Literal values
  - Variables
  - Expressions
- Variables and expressions are evaluated first
  - Determine the resulting value before invoking
  - Feed it into the function

Slides courtesy of Dr. Alan McLeod

A Function with Parameters

Here is a (useless) function that displays the larger of two numbers

```
def showHighest(num1, num2) :
  if num1 > num2 :
     print(num1, "is higher.")
  else :
     print(num2, "is higher.")
```


Slides courtesy of Dr. Alan McLeod

A Function with Parameters - Cont.

- When you invoke this (useless) function, you need to supply two things for the parameters
  - You supply two numbers as arguments

```
showHighest(3.4, 6.7)
```

- The code in **`showHighest(…)`** runs and the larger number is displayed
- Within **`showHighest(…)`**
  - **`num1`** has the value **`3.4`**
  - **`num2`** has the value **`6.7`**

Slides courtesy of Dr. Alan McLeod

A Function with Parameters - Cont.

- To put it another way …
- The positional arguments **`3.4`** and **`6.7`** have been *mapped* into the parameters **`num1`** and **`num2`**
- **`num1`** and **`num2`** are variables that have been created in the function's parameter list and are *local* to the function

Slides courtesy of Dr. Alan McLeod

Preview: Keyword Arguments

- We've usually invoked functions using *positional arguments*
  - This is the typical approach to arguments shown on the previous slides
  - *e.g.*, **`print("Hello", "Alan")`**
- We've also invoked **`print(…)`** like so

```
print("Hello", "Alan", sep="\n")
```

- The **`sep="\n"`** thing is called a *keyword argument*
  - We will learn more about keyword arguments and *default arguments* shortly

Slides courtesy of Dr. Alan McLeod

Functions Returning a Value

- A function may *return* something
  - The "something" can be any Python type
    - A `str`, an `int`, a `float`, *etc.*
- Functions that don't return anything are sometimes called *procedures*
  - Like **`print()`**, for example
- We routinely use functions that return something
  - **`input()`**
  - **`float()`**
  - **`int()`**
  - …

Slides courtesy of Dr. Alan McLeod

Functions Returning a Value – Cont.

- How can **`showHighest(…)`** be changed to return the highest number instead of printing it out?
  - It is rather tacky to have functions print things instead of returning them

```
def showHighest(num1, num2) :
  if num1 > num2 :
   return num1
  else :
   return num2
```


Slides courtesy of Dr. Alan McLeod

Functions Returning a Value - Cont.

Alternatively,

```
def showHighest(num1, num2) :
  if num1 > num2 :
   return num1
  return num2
```

This works and is more efficient!

Slides courtesy of Dr. Alan McLeod

Returning Values

- You can have as many **`return`** statements as you want in a function
- If you don't have a **`return`** statement, then your function does not return anything
  - It is invoked without expecting any value to come out of the function
    - No assignment required when invoking
- Execution of a function stops as soon as you execute the **`return`** statement, even if there is code after the **`return`** statement

Slides courtesy of Dr. Alan McLeod

Returning Multiple Values

- You can return more than one value with a single **`return`** statement!
- Consider another useless program that returns two numbers in order

```
def orderNums(num1, num2) :
    if num1 > num2 :
        return num2, num1
    return num1, num2
```


Slides courtesy of Dr. Alan McLeod

Returning Multiple Values – Cont.

- Returning multiple values is not *really* returning multiple values
- The function is actually returning a single tuple
- How can we extract the two returned values from the tuple?

```
x1, x2 = orderNums(15, 7)
print(x1, "comes before", x2)
```


Slides courtesy of Dr. Alan McLeod

The Advantages of Functions

- Each function is a building block for your program

- Construction, testing and design is easier
- Functions avoid code duplication
- Functions make re-use of your code more likely
- Well-written functions reduce the need for extensive comments

Slides courtesy of Dr. Alan McLeod

Designing a Function

- A function should only do <u>one thing</u>
  - If you describe the function and need to use the word "and", then it is probably doing more than one thing
- Try to keep the parameter list as short as possible
- The function itself should be short
  - In the range of 1 to 15 lines, ideally
  - Not larger than can be displayed on the screen
- Functions can be declared inside other functions
  - Known as *nested* functions
  - <u>Avoid</u> unless you have a good reason!

Slides courtesy of Dr. Alan McLeod

Designing a Function - Cont.

- Try to get your function to return something rather than print something
  - Trust your console I/O to a function like `main()`
- By convention, `main()` should always be the starting point of your program
- We will discuss some additional topics shortly that will make your functions easier to write and use
  - Default arguments
  - Keyword parameters
  - Raising exceptions
  - Checking argument types

Slides courtesy of Dr. Alan McLeod

Designing a Function - Cont.

- Choose descriptive function and parameter names
  - It should be obvious what the function is doing
- If you only need to add a bit more code to make your function more universally applicable – do it!
- Be prepared to re-structure a working program to get a better design
- Try to always check all your parameter values for legality
  - Later: raise an exception when they are illegal
- Add a doc string to every function except `main()`

Slides courtesy of Dr. Alan McLeod

Designing Programs With Functions

- Start with a *functional decomposition* of the problem
  - Write the function headers and add parameters
  - Use the **`pass`** statement as the body
  - Put the return value(s) in a comment initially
- Make sure each function does one thing only
- You may find a need for additional functions as you fill in the code for each function
- Don't be afraid to further decompose a function if it is getting too big or doing too many things

Slides courtesy of Dr. Alan McLeod

The Game of Nim

- Or, one of the many variations on Nim
- User plays against the computer
- Take turns removing marbles from a pile
  - May remove between 1 and half of the remaining marbles
- Winner leaves one marble in the pile
- Many values are randomly chosen
  - Number of marbles initially in the pile
  - Who get to go first
  - Number of marbles are removed by the computer

Slides courtesy of Dr. Alan McLeod

Demos - Game of Nim

- Two versions
  - Single function (all in `main()`)
  - Multi-function
- The multi-function version has more lines
- The single-function version has more indentation
- The single-function version has nested loops
  - The multi-function version does not!
- Which do you prefer?
- Which would be easier to add features to?

Slides courtesy of Dr. Alan McLeod

Testing and Debugging

- You can choose to test one function at a time
  - Add temporary code to `main()` to invoke your test function with test values
    - Display the return value(s)
  - You know that any failures are from the function currently being tested, not elsewhere
- Small functions are much easier to debug
  - It's difficult to test a single, large function

Slides courtesy of Dr. Alan McLeod

You Decide!

- Multi-function PROs
  - Easier to design
  - Easier to construct
  - Easier to read
  - Requires fewer comments
  - Easier to test and fix
  - Easier to re-use
- CONs
  - Longer
  - Slower? (not much…)

Slides courtesy of Dr. Alan McLeod

Variable Scope

- A variable created inside a function is known inside that function
  - These variables are called *local variables*
- A variable created at the same level as the function headers is known everywhere in the program
  - These variables are called *global variables*
- What do I mean by "known"?

Slides courtesy of Dr. Alan McLeod

Variable Scope – Cont.

- A variable's *scope* is the part of the program where its value can be used
  - Local variables: inside its function
    - And any other functions or statements nested in that function
  - Global variables: everywhere
- Changing the value for a global variable in a function requires an extra step
  - "Re-declare" it using the `global` keyword

Slides courtesy of Dr. Alan McLeod

Global Variables

- The problem with globals is that any function can mess with them
  - It is easy to lose track of how they are being used
- Global variables violate the principle of functional isolation!
- Two simple rules
  - Don't declare global variables unless the vast majority of your functions will use this variable
    - You must think your code will be significantly easier to work with and read as a result
  - You can declare constants as global variables
    - The constant's variable name should be in all uppercase

Slides courtesy of Dr. Alan McLeod

Keyword Arguments

- Suppose you have a function with several parameters, but you don't want to worry about supplying values in the matching order
- You can use keyword arguments to supply the arguments in any order with the syntax:

```
parameter_name = argument
```

- Demo: KeywordArguments.py

Slides courtesy of Dr. Alan McLeod

Keyword Arguments - Cont.

- All positional arguments must come before keyword arguments
- After that, the keyword arguments can be in any order
- Unless the function has default arguments you must still supply arguments for each parameter

Slides courtesy of Dr. Alan McLeod

Default Arguments

- Can make it optional for the user to supply all the arguments
  - Functions become much more flexible
- You do this by creating *default arguments* in your function definition statement
- Default arguments must come after all positional parameters
- The same syntax as for keyword arguments, but used in the `def` line instead of the invoking line
- Demo: DefaultArguments.py

Slides courtesy of Dr. Alan McLeod

Default Arguments - Cont.

- You must decide which parameters are optional
  - If any
- You must make assumptions to come up with values for those optional parameters
- Supplying an argument value for a default argument replaces the default value
- Reduces the need for multiple versions of the same function

Slides courtesy of Dr. Alan McLeod

Naming Things

- Applies to naming parameters, functions, methods and classes
- The name should reveal the intention of what it is you are naming
  - You should not need to add a comment to a variable declaration to provide further explanation
  - A comment is OK if you want to record the units of the variable or if you need to explain the initial value

Slides courtesy of Dr. Alan McLeod

Naming Things - Cont.

- Avoid Disinformation
  - Avoid using words that might have multiple meanings
- Make Meaningful Distinctions
  - Don't use artificial means of distinguishing similar names (*e.g.*, **`account0`** or **`account1`**)
- Use Pronounceable Names
  - It is easier for our brains to remember a variable name if it is pronounceable

Slides courtesy of Dr. Alan McLeod

Naming Things - Cont.

- Use Searchable Names
  - Single or even two-letter variable names will be difficult to locate using a text search
  - If a loop counter has no intrinsic meaning then it is OK to use **`i`**, **`j`** and **`k`** (but not **`l`** !!!) as loop counters
- Avoid Encodings
  - Do not use a prefix or suffix to indicate the type or membership of a variable
- Use One Word per Concept
  - For example don't use all of the terms "manager", "controller" and "driver" – what is the difference?

Slides courtesy of Dr. Alan McLeod

Misc. Style Rules

- Define `main()` at the top or end of your program
  - Don't do it in the middle!
- Use one `import` statement per line
- Put `import` statements before globals but after block comments and module-level doc strings.
- There is a big set of additional rules for doc strings themselves that I won't get into here …

Slides courtesy of Dr. Alan McLeod

Google Python Style Guide

`http://google-styleguide.googlecode.com/svn/trunk/pyguide.html`

Slides courtesy of Dr. Alan McLeod

What is an Exception?

What do you see if you try something like this?

```
print(int("Hello!"))
```

```
>>> print(int("Hello"))
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    print(int("Hello"))
ValueError: invalid literal for int() with base 10:
'Hello'
```


Slides courtesy of Dr. Alan McLeod

What is an Exception? - Cont.

- A *syntax error* is what you get when your syntax cannot be recognized by the interpreter
- All other errors occur when your code is running
  - An exception is a *run-time error*
- Every run-time error in Python has a name
  - This is the *type* of the exception
- For a list of exception types see Chapter 6 in the Python Standard Library documentation
- The example on the previous slide was a "ValueError" exception

Crash Prevention!

- Normally an exception is what you see when your program has crashed from a fatal error
- Better programs catch exceptions before this happens!
- This gives you a chance to fix the problem.
- You catch exceptions using `try-except` statements

Catching Exceptions

Syntax for a simple try-except statement

```
try :
    try_statements          ——— Give this a shot …
except Exception :          ——— but if this error occurs …
    except_statements       ——— don't crash and do this
```


Catching Exceptions - Cont.

- *`Exception`*
  - The name of the exception you are catching
- *`try_statements`*
  - A section of code that could generate a run-time error
  - The code here stops as soon as there is an error
- *`except_statements`*
  - The code that will execute if the exception is generated

Catching Exceptions - Cont.

- How do you know which exception(s) to catch?
  - There's no easy answer to this question
- Observe the error generated and get the name of the exception from the traceback listing
- … or look in the Python docs

Slides courtesy of Dr. Alan McLeod

Catching More Than One Exception

- Suppose your code could generate more than one kind of exception?
- You can be prepared to catch more than one!

```
try :
  try_statements
except Exception1:
  except_statements
except Exception2 :
  except_statements
…
```


Slides courtesy of Dr. Alan McLeod

Demo: Robust Input

- Until now, we have had to assume the user enters a number when we tell him to
  - Very trusting of us …
- Now we don't have to
  - He or she can be an idiot and our input won't crash
- Demo: MoreRobust.py

Slides courtesy of Dr. Alan McLeod

Prevent or Catch?

- Sometimes it should not be necessary to catch exceptions
- You should prevent them from happening in the first place by using preventative code
- Demo: WhichIsBetter.py

Slides courtesy of Dr. Alan McLeod