CISC101 Reminders & Notes

- Assignment 2 grades are posted in Moodle
- Test 2 is marked
  - Grades will be posted in Moodle
  - Tests will be handed back in tutorial this week
- Assignment 3 is now posted
  - Due on Sunday, March 20th
- May have a guest lecture ...
  - Notes will **not** be posted on the website
  - Related questions **will** be on the exam

Slides courtesy of Dr. Alan McLeod

Today

- Cover material on exceptions from last lecture
  - Slides 52-60
- Continue with exceptions
- Strings
  - What we already know
  - Keywords and BIFs
  - Methods (lots of them!)
  - Demos
- Basic file input and output

Slides courtesy of Dr. Alan McLeod

`None` – What is it and Why is it Useful?

- **`None`** is a built-in constant
  - Indicates the absence of a value (*i.e.*, nothing)
- **`None`** is **not** zero
  - Zero is a number, **`None`** is not
- Use it when you need a value but don't have one
  - Return it if you can't return something meaningful
  - Use it to create a variable for which you have no value
  - Use it for default arguments for which there are no sensible values to assign
- You can test to see if something equals **`None`** in a boolean expression

Slides courtesy of Dr. Alan McLeod

Demo: Robust Input Between Limits

- Modify `getInt(…)` from MoreRobust.py
  - Can supply limits for the integer number
- What if you don't want to use one or both limits?
  - Use default arguments!
- What would be a good default limit?
  - We don't want to assign an inappropriate limit …
  - Solution: use **`None`** !
- Demo: MoreRobustRange.py

Slides courtesy of Dr. Alan McLeod

Raising Exceptions

- What do you do when your function cannot do its job?
- You could return something so the invoking function knows that there is a problem
- Or, you could raise an exception
  - This is better in many situations

Slides courtesy of Dr. Alan McLeod

Raising Exceptions - Cont.

- You can raise (or "throw") an exception by using the **`raise`** keyword

```
raise exception_name
```

- If you want to supply a "reason" as well,

```
raise exception_name(reason_string)
```


Slides courtesy of Dr. Alan McLeod

Raising Exceptions - Cont.

- This is just like creating an error condition, but in an artificial way …
- Demo: RaiseException.py
- Demo: RaiseExceptionWithMessage.py

Slides courtesy of Dr. Alan McLeod

Raising Exceptions - Cont.

- Whenever you raise an exception, the function that raised the exception is halted
  - No other code in the function will execute
- If the function call was part of an expression then the rest of the expression will not be evaluated
- As you know – if the exception is not caught, you will see the nasty red stuff!

Slides courtesy of Dr. Alan McLeod

Raising Exceptions - Cont.

- It is easiest to just raise one of the existing exceptions
- If it is because of a parameter error then `ValueError` is appropriate
- Creating our own exception objects is beyond the scope of this course

Slides courtesy of Dr. Alan McLeod

Strings

- String manipulation is a frequent activity in any programming language
  - Speech analysis
  - Text searching and indexing
  - Spell and grammar checkers
  - Program (code) interpretation
  - Scanning emails for SPAM
  - … and more
- A string is a kind of data structure
  - Like a tuple, but they have lots of methods
    - Tuples have very few methods

Slides courtesy of Dr. Alan McLeod

Strings Thus Far

- String literals

```
"Hello there! “
‘CISC 101‘
"""Multiline
string"""
```

- You can store a string in a variable
  - Just like anything else
- They are of type `str`
- `str(…)`is a BIF
  - Returns a string version of the given argument

Slides courtesy of Dr. Alan McLeod

Strings Thus Far – Cont.

- `input(…)` is a BIF
  - Returns user input from the keyboard as a string
- You can use escape sequences
  - `\n`, `\”`, `\’`, `\\`, `\t`
  - These control how a string is displayed
- The string `format()` method is useful to format numeric output for display

Slides courtesy of Dr. Alan McLeod

Strings Thus Far – Cont.

- You can concatenate strings using +
- You can generate repeating strings using *
- You can compare strings
  - Use ==, >, <, >=, <= and !=
  - Just like comparing numbers, but you must have a string on both sides of the operator
  - Strings are compared on the basis of the ASCII code values for their individual characters
- They are a type of collection
  - A collection a characters

Slides courtesy of Dr. Alan McLeod

Strings And Collections - Similarities

- Many collection "accessories" work with strings
  - The slice operator **`[ : ]`**
  - **`in`** and **`not in`**
    - A string must be placed on both sides
  - **`for`** loops
  - **`len(…)`** BIF
  - **`list(…)`** and **`tuple(…)`** BIFs
    - Create a list or a tuple with the individual characters
  - **`sorted()`** BIF
    - Returns a sorted list of individual characters
  - **`reversed()`** and **`enumerate(…)`** BIFs
- Demo: StringsAsCollections.py

Slides courtesy of Dr. Alan McLeod

Strings And Collections - Differences

- Strings are immutable
  - Cannot put the slice operator on the left side of the assignment operator
  - **`del`** does **not** work
- Tuples only have two methods
  - **`count(…)`** and **`index(…)`**
  - Strings have many methods …

Slides courtesy of Dr. Alan McLeod

Other String BIFs – Unicode and ASCII

- **`chr(…)`**
  - Takes an integer argument (a Unicode value)
    - For the "narrow build" of Python that we use, this number can range from 0x0000 to 0xFFFF
      - 2 bytes, compared to ASCII's 1 byte
      - But ASCII values still work!
  - Returns the corresponding character
- **`ord(…)`** does the reverse of **`chr(…)`**
  - Takes a single character as a string for the argument
  - Returns the character's code value
- Demo: ASCIITable.py

Slides courtesy of Dr. Alan McLeod

ASCII Characters - Observations

- The empty boxes are non-printing characters
  - They do something like <enter> or <del> or <\n>, *etc*.
- Some characters seem to have a <backspace> built in …
- ASCII 32 is a space
- ASCII 10 must be an <enter> (or "newline")
- Keyboard characters stop at ASCII 127
- Characters from ASCII 128 to 254 are called "extended characters"
  - Not all of them are available in the console window

Slides courtesy of Dr. Alan McLeod

Unicode Characters

- Demo: UnicodeTable.py
  - Unicode numeric values are not displayed and only a fraction of the table is printed out
  - Most empty boxes represent un-assigned Unicode values
    - They really are empty
- Demo: UnicodeBox.py

Slides courtesy of Dr. Alan McLeod

Character Codes for Escape Characters

- Demo: EscapeCharacters.py
  - Uses the **`ord(…)`** BIF
- You now know that you could use the **`chr(…)`** BIF to generate these escape sequences …

Slides courtesy of Dr. Alan McLeod

String Methods

- Just like a list, a string has many methods
  - 35 of them (a subset) are listed here
- The next 3 slides list the methods in alphabetical order
  - There is no other description
  - Note the use of default arguments
- Remember that they are invoked as follows:

```
string_variable.method_name()
```


Slides courtesy of Dr. Alan McLeod

```
aString.capitalize()
aString.center(width)
aString.count(str, beg=0, end=len(aString))
aString.endswith(obj, beg=0, end=len(aString))
aString.expandtabs(tabsize=8)
aString.find(str, beg=0, end=len(aString))
aString.format(args)
aString.index(str, beg=0, end=len(aString))
aString.isalnum()
aString.isalpha()
aString.isdigit()
aString.islower()
```



```
aString.isspace()
aString.istitle()
aString.isupper()
aString.join(seq)
aString.ljust(width)
aString.lower()
aString.lstrip()
aString.partition(str)
aString.replace(str1,str2,num=aString.count(str1))
aString.rfind(str, beg=0, end=len(aString))
aString.rindex(str, beg=0, end=len(aString))
aString.rjust(width)
```



```
aString.rpartition(str)
aString.rstrip()
aString.split(str=" ", num=aString.count(str))
aString.splitlines(num=aString.count('\n'))
aString.startswith(obj, beg=0, end=len(aString))
aString.strip()
aString.swapcase()
aString.title()
aString.translate(str, del="")
aString.upper()
aString.zfill(width)
```


String Methods - Cont.

- Each method returns something
- None of them alter the *aString* object
  - Strings are immutable!
- Categorize by return value:
  - boolean (`True` or `False`)
  - integer
  - another string
  - a list or tuple of strings

Boolean Returns

***aString*.endswith(*obj*, *beg*=0, *end*=len(*aString*))**

- Returns `True` if *`aString`* has *`obj`* at the end of the string or `False` otherwise
- *`obj`* is usually a string, but can be a tuple of strings
  - Returns `True` if any one of the strings match
- You have the option of limiting the search to a portion of *`aString`*

***aString*.startswith(*obj*, *beg*=0, *end*=len(*aString*))**

- Just like `endswith(…)`, but looks at the start of *`aString`* instead

Slides courtesy of Dr. Alan McLeod

Boolean Returns - the "is" Ones

***aString*.isalnum()**

- Returns `True` if all of the characters in *`aString`* are alphanumeric (letters and numbers only), `False` otherwise

***aString*.isalpha()**	`True` if all alphabetic (letters only)
***aString*.isdigit()**	`True` if all digits (numbers only)
***aString*.islower()**	`True` if all letters are lowercase
***aString*.isspace()**	`True` if only whitespace(tabs, *etc.*)
***aString*.istitle()**	`True` if "titlecased"
***aString*.isupper()**	`True` if letters are all uppercase

Slides courtesy of Dr. Alan McLeod

Aside - Titlecase

- What is titlecase?
  - All words in the string must start with a capital letter and all other letters are lower case

***aString*.title()**

- Returns a copy of *`aString`* in titlecase

Slides courtesy of Dr. Alan McLeod

Integer Returns

***aString*.count(*str*, *beg*=0, *end*=len(*aString*))**

- Returns a count of how many times *`str`* occurs in *`aString`*, or a substring of *`aString`* as specified by *`beg`* and *`end`*

Slides courtesy of Dr. Alan McLeod

Integer Returns - Cont.

`aString.find(str, beg=0, end=len(aString))`
`aString.index(str, beg=0, end=len(aString))`

- Returns the location of the first occurrence of *`str`* in *`aString`*
- Starts the search from the beginning of the string, or searches a substring specified by *`beg`* and *`end`*
- `find(…)` returns -1 if not found, `index(…)` raises an exception if not found

`aString.rfind(str, beg=0, end=len(aString))`
`aString.rindex(str, beg=0, end=len(aString))`

- Same as above but searches *`aString`* from the end

Slides courtesy of Dr. Alan McLeod

String Returns

`aString.capitalize()`

- Returns a string that is the same as *`aString`* except the first letter is capitalized

`aString.lower()`

- Returns a string that has all the upper case letters in *`aString`* converted to lower case

`aString.swapcase()`

- Returns a string with the case of all letters in *`aString`* inverted

Slides courtesy of Dr. Alan McLeod

String Returns - Cont.

`aString.upper()`

- Returns a string with all lower case letters in *`aString`* switched to uppercase

`aString.center(width)`

- Returns a string with spaces added to *`aString`* to centre it in a column of size *`width`*

`aString.ljust(width)`

- Returns a string with spaces added to *`aString`* to left justify it in a column of size *`width`*

Slides courtesy of Dr. Alan McLeod

String Returns - Cont.

`aString.rjust(width)`

- Returns a string with spaces added to *`aString`* to right justify it in a column of size *`width`*

`aString.expandtabs(tabsize=8)`

- Returns a version of *`aString`* that has all the tab characters converted to spaces
  - The default is 8 spaces per tab

`aString.join(seq)`

- Joins all string representations of the elements in the list *`seq`* together using *`aString`* as the separator

Slides courtesy of Dr. Alan McLeod

String Returns - Cont.

***aString*.lstrip()**

– Removes all leading whitespace (tabs, spaces, linefeeds, etc.) in *aString*

***aString*.rstrip()**

– Removes all trailing whitespace in *aString*

***aString*.strip()**

– Removes all leading and trailing whitespace in *aString*

Slides courtesy of Dr. Alan McLeod

String Returns - Cont.

***aString*.replace(*str1*, *str2*, *num*=*aString*.count(*str1*))**

– Replaces all (or *num*) occurrences of *str1* in *aString* with *str2*

***aString*.format(*args*)**

– We've used this one already

– The supplied arguments are formatted according to the "replacement fields" contained in the string itself

Slides courtesy of Dr. Alan McLeod

Tuple or List Returns

***aString*.partition(*str*)**

– Carries out a `find(…)` and then splits *aString* into a tuple of three strings - the stuff before *str*, *str* itself and all the stuff after *str*

***aString*.rpartition(*str*)**

– The same as `partition(…)`, but it searches from the end instead

Slides courtesy of Dr. Alan McLeod

Tuple or List Returns - Cont.

***aString*.split(*str*=" ", *num*=*aString*.count(*str*))**

– Returns a list of strings parsed out of *aString* using *str* as a delimiter

– *num* can specify a maximum size to the list

– If *str* is not supplied, a `strip(…)` is applied and then whitespace is used as a delimiter

***aString*.splitlines(*num*=*aString*.count('\n'))**

– Splits *aString* and returns a list using the newline character as a delimiter

– All newline characters are removed

Slides courtesy of Dr. Alan McLeod

Demo: String Methods

- Start with the tuple or list returns methods
- Looks at how you can analyze larger amounts of text
- Demo: StringMethods.py

Slides courtesy of Dr. Alan McLeod

File I/O

- Files provide a convenient way to store and re-store to memory larger amounts of data
- Use a data structure like a list to store the data in memory
- Three kinds of file I/O
  - Text
  - Binary
  - Random access
- We will stick with text I/O in this course
- Text files can be read by Notepad, for example

Slides courtesy of Dr. Alan McLeod

Text File Output

file_variable **= open(*filename, mode*)**

- ***filename*** is the name of a file
  - Must be in the same folder as your program
  - It is a string
- ***mode*** is also a string
  - **'r'** for reading only
  - **'w'** for writing only
  - **'a'** for appending to a file
- The default mode is **'r'**

Slides courtesy of Dr. Alan McLeod

Aside - Other File Modes

- **r+** - read and write (same as **w+** or **a+**)
- **rb** - binary read
- **wb** - binary write
- **ab** - binary append
- **rb+** - binary read and write (same as **wb+** and **ab+**)
- If you are having problems with line terminators, you can also try **rU**
  - Read with "universal newline support"

Slides courtesy of Dr. Alan McLeod

Text File Output - Cont.

- If you open an existing file for writing using mode **'w'**, the old file will be overwritten with a new file
  - All the old contents will be lost
- If you want to add to an existing file without erasing the old contents, use the **'a'** mode
  - **'a'** for append
- If you do not provide a path, the file is created in the same folder as your program

Slides courtesy of Dr. Alan McLeod

Text File Output - Cont.

- To write information to a file, use the **write()** method

```
file_variable.write(a_string)
```

- The **write()** method does not add a line terminator to the end of the string
  - If you want to write a line, and have the next output go to the next line, you need something like this

```
file_variable.write(a_string + '\n')
```


Slides courtesy of Dr. Alan McLeod

Text File Output - Cont.

- Once you are finished writing to the file don't forget to close the file using

```
file_variable.close()
```

- If you don't do this, you run the risk of leaving a corrupted file on your hard disk!

Slides courtesy of Dr. Alan McLeod

Sequential File Access

- Text file I/O uses sequential access
- Think of having a little "pointer" in the file marking the end of what you have read
- As you read (or write), the pointer moves ahead
- The pointer cannot move backwards

- The only way to re-read something is to close the file and open it again
  - This moves the "pointer" back to the beginning

Slides courtesy of Dr. Alan McLeod

vs. Random File Access

- This can be used only with binary files
- Seeks a certain byte location in the file
  - You must know the exact structure of the file to do this
- Read or write data from this location
- Seek again …

Slides courtesy of Dr. Alan McLeod

Text File Input

- Use the **`open(…)`** method as shown on slide 35
- Use the **`readline()`** method to read a line up to and including a linefeed character
  - This method returns a string
- You might wish to use something like **`rstrip()`** on the string to remove the linefeed, and any other whitespace at the end of the string

Slides courtesy of Dr. Alan McLeod

Text File Input - Cont.

- There are other file reading methods
- **`read()`**
  - Reads the entire file and returns a single string
- **`readlines()`**
  - Reads the entire file and returns a list of lines of text

Slides courtesy of Dr. Alan McLeod

Text File Input - Cont.

- Invoke the **`close()`** method when you are done reading
- A **`for`** loop can simplify input …
- Demo: TextFileIO.py

Slides courtesy of Dr. Alan McLeod