CISC101 Reminders & Notes

- Assignment 2 is due on Sunday at 11:59PM

- There are no CISC 101 lectures, tutorials or labs during Reading Week

- Test 2 will occur the week after Reading Week

Today

- Looping through strings
- The `for` loop
- Built-in collections or *data structures*
  - Tuples
  - Lists
- The slice operator
- More on programming style
  - Some review
  - Some new material

Looping Through Strings

- Strings are actually a kind of *collection* in Python
  - A collection of characters - makes sense!
- The `len(aStr)` BIF returns the length of a string
  - Or any other collection
- Use the *slice* operator [] to access any character

```
aString[index]
```

  - `index` ranges from 0 (first character on the left) to `len(aString) - 1` (last character on the right)

- Demo: IterateString.py

The `for` Loop

```
for variable_name in iterable :
 line1
 line2
 …
```

- You make up ***`variable_name`***

- ***`iterable`*** is a collection, such as a string

The `for` Loop - Cont.

- A `for` loop can be replaced with a `while` loop
- These two loops do exactly the same thing:

```
i = 0
testString = "Happy Reading Week!"
```

```
while i < len(testString) :
    print(testString[i])
    i = i + 1
```

```
for aChar in testString :
    print(aChar)
```

The `for` Loop - Cont.

- **But** the `for` loop is easier to use with collections
  - Goes through each element in order
  - No indicies
  - No need to call the `len(…)` BIF

Data Structures

- What is a *data structure*, anyways?

> In practical terms, it is a variable that is capable of holding more than a single value

- Python has four built-in data structure types:
  - Lists
  - Tuples
  - Sets
  - Dictionaries

Focus on these ones.

- Strings are really just a kind of tuple

Lists *vs*. Tuples

- A list is a set of items enclosed in `[ ]`
- A tuple is a set of items enclosed in `( )`
- You **can** change the items within a list and its length at any time after you have created it
  - Lists in Python are mutable
- You **cannot** change the items in a tuple or change its length after you have created it
  - Tuples in Python are immutable (like "read-only")

Lists *vs*. Tuples – Cont.

- Numbers and strings are also immutable
  - You can’t mess with the individual digits of a number or the individual characters of a string after you have created them
  - You can only re-assign variables that are numeric or string types
  - *Don’t believe me? Let’s try using the slice operator to try to change a character in a string* …

Slides courtesy of Dr. Alan McLeod

Dictionaries

- Dictionaries or “dicts” are enclosed in **`{ }`**
- They consist of `key : value` associations
- For example,

```
cisc101Dict = {‘instructor’ : ‘SJW’, \
               ‘room’ : ‘BIO1203’, \
               ‘exclusion’ : ‘CISC110’ }
```

- We will look at these more closely later …

Slides courtesy of Dr. Alan McLeod

Sets

- Are new to Python 3
- Items enclosed in **`{ }`** (like dictionaries)
- Each item **must** be unique
  - If you try to create a set with duplicate items, the duplicates will be discarded
- We will look at these more closely later too …

Slides courtesy of Dr. Alan McLeod

Lists

- Lists can contain items of all the same type

```
[3, 2, -1, 10]
```

- Lists can also contain a mixture of types

```
[4.2, ‘7abc’, 3, aVar]
```

- Lists can store variables as well as literals!
- All *elements* are comma-separated

Slides courtesy of Dr. Alan McLeod

Tuples

- Can store a mixture of types, just like lists

  `aTuple = (4, 3.2, ‘abc’, 7, -3, ‘ding’)`

- Since a tuple is immutable, you cannot change its values
  - You can’t do anything like `aTuple[1] = 7`
- Use (*element*,) to create a single-element tuple
  - Python needs the comma
- Use `()` to create an empty tuple

Empty Lists

- You can create an empty list like so:

  `mtList = []`

- You can add and alter the values in a list later
  - Lists are mutable, unlike tuples
- Useful things:
  - The slice operator
  - The + operator
  - The `append(`*anElement*`)` function

Slice Operator

- You can extract single elements or a set of elements from a collection using the *slice* operator:

  [*index*] **or** [*start_index* : *end_index*]

  - All indicies are `int` numbers
- Locations are *indexed* from 0 (first element)
  - Maximum index is *len(collection*) `- 1` (last element)
- The slice operator with the `:` returns a *range* of elements
  - No : returns a single element

Slice Operator - Cont.

- When using [*start_index* : *end_index*], you can supply one or two numbers
- Omit *start_index* ?
  - The slice starts at the start of the collection
- Omit *end_index* ?
  - The slice ends at the end of the collection.
- Use both *start_index* and *end_index*?
  - Slice starts at *start_index*
  - Slice ends at *end_index* `- 1`

Slice Operator - Cont.

- If `end_index` is too large, then the slice defaults to the end of the list
- The slice operator can be used on either side of an assignment operator!
- You can also number the elements backwards, where -1 is the last number in the list …
- Let’s try a few out at the prompt!

Slice Operator Examples

```
>>> test = [2, 1, 3, -1, 4, 6]
>>> test[3]
-1
>>> test[-1]
6
>>> test[4 :]
[4, 6]
>>> test[ : 3]
[2, 1, 3]
>>> test[1 : 3]
[1, 3]
```

Slice Operator Examples – Cont.

```
>>> test[1 : 3] = [10, 30]
>>> test
[2, 10, 30, -1, 4, 6]
>>> test[-1] = 600
>>> test
[2, 10, 30, -1, 4, 600]
```

Other Operators For Lists and Sets

- What is there in addition to the slice operator?
- + can be used to concatenate lists
  - Requires a list on both sides or a tuple on both sides
  - You **cannot** mix types!
- `*` is used to generate multiples of lists
  - Must have an `int` after the `*`
  - Works with tuples or lists
    - Remember `“abc” * 3  = “abcabcabc”`?

Other Operator Examples

```
>>> test
[2, 10, 30, -1, 4, 600]
>>> testTwo = [5, 10, 15]
>>> test + testTwo
[2, 10, 30, -1, 4, 600, 5, 10, 15]
>>> testTwo * 3
[5, 10, 15, 5, 10, 15, 5, 10, 15]
```

Keywords Used with Lists

- del *aList[anIndex]*
- del *aList[startIndex* : *endIndex]*
  - Deletes the element(s) from list *aList*
  - The slice operator specifies the element to delete
- *element* in *aCollection*
- *element* not in *aCollection*
  - Determine if *element* is in a list or tuple (or not)
  - Return True or False
- for *variableName* in *aList*

Keyword Examples

```
>>> test
[2, 10, 30, -1, 4, 600]
>>> del test[3]
>>> test
[2, 10, 30, 4, 600]
>>> del test[1 : 3]
>>> test
[2, 4, 600]
>>> 4 in test
True
>>> 100 in test
False
>>> 100 not in test
True
```

for Loop Example

```
>>> test
[2, 4, 600]
>>> for aNum in test :
  print(aNum, end=', ')

2, 4, 600,
```

Some Built-In Functions for Lists

- `len(`*`aCollection`*`)`
  - Returns the number of elements in the collection
- `list(`*`iterable`*`)`
  `tuple(`*`iterable`*`)`
  - Returns a new list/tuple with the same elements
- `range(`*`start, stop, step`*`)`
  - Returns an *iterable* with integers
    - Starts with integer `start` (optional parameter)
    - Stops at `stop – 1`
    - Increases integers by `step` (optional parameter)
  - Often used with a `for` loop …

Slides courtesy of Dr. Alan McLeod

The `range()` BIF

- This function returns an *iterable*, not a list
- Where did we see iterable before?
  - An iterable is a collection, such as a string
- Let’s create one and display its contents
- Can convert to a list or tuple
  - Use `list(`*`iterable`*`)` or `tuple(`*`iterable`*`)`

Slides courtesy of Dr. Alan McLeod

The `range()` BIF - Cont.

- *Say, that’s handy!*
- For example, these two loops are the same:

```
i = 0
while i < 20 :
   print(i)
   i = i + 1
```

```
for i in range(20) :
   print(i)
```

Slides courtesy of Dr. Alan McLeod

`sorted(…)` and `reversed(…)` BIFs

- `sorted(`*`iterable`*`)`
  - Returns a sorted version of *`iterable`*
  - Does not change *`iterable`*!
- *`aList`*`.sort()`
  - Sorts *`aList`* ”in situ”, changing it
- `reversed(`*`iterable`*`)`
  - Often used with a `for` loop …
  - Reverses the direction of iteration
    - Starts at the last element and ends with the the first

Slides courtesy of Dr. Alan McLeod

enumerate(…) and zip(…) BIFs

- for *i*, *element* in enumerate(*iterable*):
  - Provides an index number and an element for collections
- for *e1*, *e2*, … in zip(*iter1*, *iter2*, …):
  - Provides a way to loop through any number of collections at the same time
- Demo: ListBIF.py

Slides courtesy of Dr. Alan McLeod

List Methods

- These methods belong to a list <u>object</u>
- *list* is the name of a list; *obj* is a value

list.append(*obj*)	# appends *obj* to list
list.count(*obj*)	# counts occurrences of *obj*
list.index(*obj*)	# first occurrence of *obj*
list.index(*obj* , *i*, *j*)	# search between *i* and *j*
list.insert(*index*, *obj*)	# insert *obj* at *index*
list.pop()	# removes the last element
list.remove(*obj*)	# search for and remove *obj*
list.reverse()	# reverses in place
list.sort()	# sorts in place

Slides courtesy of Dr. Alan McLeod

List Methods - Cont.

- None of these methods work for tuples
  - They only work on lists
- Consult the Python Tutorial, Chapter 5 for more information on data structures
- Demo: ListMethods.py

Slides courtesy of Dr. Alan McLeod

Methods *vs*. BIFs

- A method belongs to an *object*
  - Objects are data structures like strings or lists
    - More complicated than numbers or Booleans
  - Need an *instance* of an object to call the method on
    - *e.g.*, *aString*.format(…), *aList*.pop(), *etc.*
  - Invoke methods using *an_object*.*method_name*(…)
- A BIF does not belong to any object
  - Can just call the function
    - e.g., print(…), input(…), *etc.*
  - Invoke functions using *function_name*(…)

Slides courtesy of Dr. Alan McLeod

List Method Examples

```
>>> test = [4, 5, 2, 7, 9]
>>> test
[4, 5, 2, 7, 9]
>>> test.append(12)
>>> test
[4, 5, 2, 7, 9, 12]
>>> test.pop()
12
>>> test
[4, 5, 2, 7, 9]
>>> test.pop()
9
>>> test
[4, 5, 2, 7]
>>> test.insert(2, 12)
>>> test
[4, 5, 12, 2, 7]
```

Slides courtesy of Dr. Alan McLeod

List Method Examples - Cont.

```
>>> test
[4, 5, 12, 2, 7]
>>> test.append(12)
>>> test
[4, 5, 12, 2, 7, 12]
>>> test.remove(12)
>>> test
[4, 5, 2, 7, 12]
>>> test.reverse()
>>> test
[12, 7, 2, 5, 4]
>>> test.sort()
>>> test
[2, 4, 5, 7, 12]
```

Slides courtesy of Dr. Alan McLeod

Programming Style & Documentation

- Purpose is to make your code readable and “debuggable” by you or another programmer

“*Code is read more often than it is written.*”
(Guido van Rossum)

- Internal style elements
  - Documentation (comments)
  - Spacing
  - Descriptive variable names
- Select your conventions and **be consistent**

Slides courtesy of Dr. Alan McLeod

Comments

- Add a comment at the top of your program and at the beginning of each function describing …
  - the overall purpose of the program or function
  - the main algorithm used
  - author and date created
  - any assumptions made and/or bugs found
- Function comments should state …
  - what parameters are expected by the function
  - what the function returns, if anything
  - any assumptions made about the arguments

Slides courtesy of Dr. Alan McLeod

Comments – Cont.

- When the name of a variable is not self-explanatory, add an inline comment when it is first initialized
- Add comments at the start of logical blocks
  - Indent comment same as start of block
- You don’t need to explain code that is obvious
  - Focus on code that is tricky to understand
    - *Maybe it needs to be re-written?*
- **`# TODO`** comments can be used to mark where more work is needed

Documentation Strings

- We’ve seen these already
- If you describe your function in a doc string you don’t need as much in its comment
- What would you <u>not</u> include in a doc string?
  - Author(s), date/revision number, code history, problem areas, incomplete section(s), license/copyright, *etc.*
- Write doc strings for each function in a program unless they are short and obvious
- Don’t forget that doc strings are available through the use of the `help()` BIF at the prompt

Spacing

- Use 4 spaces for indentation
- <u>Don’t mix tabs and spaces</u>
  - Not a problem if you are only using IDLE
    - When you hit the <tab> key you automatically get 4 spaces
- Long lines:
  - Keep lines < 80 characters in length
  - Use the Python continuation character \
    - Indent a continued line so that it lines up nicely
  - Break a line after a binary operator, not before

Spacing - Cont.

- Continuation examples:

```
longAssignment = aLongName + anotherLongName - \
                 anotherVariable * 2.0

returnedVal = functionCall(param1, anotherParam, \
                           param2, param3)
```

- Don’t put multiple lines of code on a single line:

```
if bingo < 3 : bork = try + again
else : we = are + all + winners
```

Spacing - Cont.

- Use one blank line above a **`def`** statement
  - No blank lines below
- A blank line inside a function can be used to delineate a block of code
  - Don't put too many blank lines inside a function
  - <u>Don't</u> double space your code!
- Put a blank line under a doc string

Slides courtesy of Dr. Alan McLeod

Spacing - Cont.

- Put a space on both sides of a **`:`**
  - Google style says no space on the left of a **`:`** …
- Put a space after a comma, but not before
- Put spaces on <u>both</u> sides of a binary operator
- Put spaces on both sides of keywords like **`in`**, **`not in`**, **`is`**, **`and`**, **`or`**, **`not`**
- <u>Do not</u> put a space after a unary operator
- <u>Do not</u> use spaces around a **`=`** when used in a function's parameter list (default and keyword arguments)

Slides courtesy of Dr. Alan McLeod

Spacing - Cont.

- No space before or after **`(`** and **`)`** unless an operator comes before or after the brackets
- Same rules for **`[ ]`** and **`{ }`**

Slides courtesy of Dr. Alan McLeod

Using Round Brackets

- Use round brackets when they are necessary
- Brackets are totally unnecessary in these cases:

```
if not(x):
if ((x < 3) and (not y)):
return (foo)
for (x, y) in dict.items():
```

- Brackets are unnecessary (but OK to have) in these cases:

```
if (x > 2):
while (x < 3):
```

- Understanding precedence will help!

Slides courtesy of Dr. Alan McLeod

Round Brackets and Tuples

- On the last slide the **`(x, y)`** is a tuple made from variables **`x`** and **`y`**

- Any list of variables separated by commas is automatically a tuple
  - You don’t need the brackets to make one

- However, if you wish to keep the brackets as a personal preference then do so

Do

```
# This program is used to demonstrate better style.
# Version 1, by Alan McLeod, 27 Oct. 2009

def product(num1, num2) :
'''This is a useless little function that does not do
  much'''

    print('Hello')
    return num1 * num2

def main() :
'''main invokes product and then waves goodbye!'''

    print product(3, 4)
    print('Goodbye!')

main()
```

Don’t!

```
def m(ll1,l1):print('Hello');return(ll1*l1);
def main():print(m(3,4));print("Goodbye!")
main  ()
```

- This works, but is incomprehensible
- How many things are wrong with this code?

Above All Else,

Be Consistent!