Contents

Tips, Tricks, and Pitfalls

Level 1

I: Strings

1. String variables are variables that contain one or more keyboard characters. You can create a string variable by enclosing in single quotes:

   ```
   'This is a single quote string'
   ```

   Single quotes are the most common way to specify a string in Python. You can also use double quotes:

   ```
   "This is a double quote string"
   ```

   Single quotes are preferred, but the most important thing is to be self-consistent.

   Python provides several 'escape characters' that you can insert into your strings. For example, if you wish to have a newline or tab in your string, you can specify them using escape characters. A few examples below (there are additional ones, but less important):

   ```
   'This string spans \ntwo lines' #'\n' specifies newline
   'This string contains \ta tab' #'\t' specifies tab
   'To specify a quotation mark in your string, it must be \'escaped\''
   'You can specify a single slash in your string by using \\'
   ```

   You can also use triple-quotes to specify a string to appear exactly as-formatted.
   This is very rare to use, except for doc strings (see the **Code Comments** section). Example:

   ```
   """This string will appear exactly as   it is
   formatted within the triple-quotes.
       !!!
   """
   ```

2. Strings can be combined using the **+** operator. Strings may be combined with other explicit strings and/or variables which contain strings. Note that non-string variables (i.e., numbers) will need to be converted to **str** prior to combining it with a string (otherwise, you'll see an error). A couple examples:

```
'This is a ' + 'combined string' + ' made up of three'

name = 'Gary'
age = 25
name + ' is' + str(age) + ' years old'
```

As you can see, combining strings with one or more variables can be unwieldy. There are better approaches to string combining, formatting, and manipulation, which will be covered in Level 4. For now, you should use the above technique.

3. Strings can be multiplied (not that useful in practice, generally):

```
>>> s = 'Hello!'
>>> print(s)
Hello!
>>> print(s*5)
Hello!Hello!Hello!Hello!Hello!
```

4. The number 1 (**int** type) is internally distinct from the string '1'. Without going into CS detail, the number 1 has an ASCII value of 1 whereas the string '1' (which is the keyboard character) has an ASCII value of 49. This applies to all numbers.

   The numbers are useful for mathematical operations, whereas the string representations are useful for strings (i.e., console output).

   The **str** conversion of a number variable (mentioned in the previous point) converts the numeric representation of the number to its ASCII (string) representation, which enables it to be combined with other strings.

   See https://en.wikipedia.org/wiki/ASCII for more details.

5. Note that there are two ways to display a variable when using the Python Console. You can either just type the variable name and press enter, or use the **print** function to print out the variable's value. There is actually a key difference between these two approaches: The former displays the Python *representation* of the variable whereas the latter displays the variable's value. This difference becomes evident when displaying strings: The former approach will display the string with the actual escape characters whereas the latter will format the string as specified.

```
>>> s = 'This contains a two\nline string'
>>> s
'This contains a two\nline string'
>>> print(s)
This contains a two
line string
```

Note that this is only relevant when using the Console. For a script, only **print** will work to output values.

II: If Statements and Booleans

1. A *Boolean* (**bool** type) variable can have one of two states: True or False. A *Boolean* expression or statement is one that evaluates if something is True or False. For example:

```
age = 50
print(age < 60) # prints True

r = age > 75
print(r) # prints False
print(type(r)) # Prints <type 'bool'>
```

   An **if** statement simply evaluates a Boolean expression for a True or False value. If the value is True, the **if** block executes; if it's False, the **if** block does not execute (though the **else** block, if specified, will execute).

2. When writing an **if** statement, it is possible to specify two or more conditions by nesting and/or using **and/or**. See a few examples below:

```
# Using boolean 'and'. Can use && instead of 'and'
if age < 30 and age > 5:
    print('You are between 5 and 30 years old')

# Using boolean 'or'. Can user || instead of 'or'
if age <= 30 or age > 50:
    print('You are either younger than 31, or older than 50')

# Combining multiple boolean statements.
if age < 30 or (age > 50 and age < 100):
    print('You are either younger than 30 or between 50 and 100 years old')

# Nesting if statements. Sometimes this is better than using 'and'.
# In this example, an 'and' would work for if age < 30 and age > 5, but then the
# else statement would be on both of those instead of each one individually.
# Nesting provides more granularity in your expression, but should only be used when necessary.
if age < 30:
    if age > 5:
        print('You are between 5 and 30 years old')
    else:
        print('You are less than 30 years old')
elif age < 50:
    print('You are between 30 and 40 years old')
else:
    print('You are at least 40 years old')
```

3. We saw common Boolean operators in the lecture (==, !=, <, >, <=, >=). There are another two in Python -- **is** and **not**:
    - The **is** operator is very similar to ==, but there is a key difference: == checks for equivalence whereas **is** checks if a variable *is* the exact same variable as another. Equivalence does not require that the variables are one and the same (**is** can be thought of as a special case of ==). For example:

```
list1 = [1, 2, 3, 4]
list2 = [1, 2, 3, 4]
list3 = list1

print(list1 == list2) # Prints True
print(list1 is list2) # Prints False
print(list3 is list2) # Prints True, due to the non-copying nature of Python assignments.
```

    - The **not** operator enables negation of a Boolean expression. For example:

```
age = 45
print(age <= 50) # prints True
print(not age <= 50) # Prints False
print(age <= 50 and age > 45) # Prints False
print(not (age <= 50 and age > 45)) # Prints True
```

4. An **if** statement on a Boolean variable may be written is several ways. For example:

```
isFemale = True
print(isFemale is True) # Prints True, but the 'is True' is redundant.
print(isFemale) # Much better
print(isFemale is not True) # Prints False, but the 'is not True' is redundant.
print(not isFemale is True) # Prints False, but the ' is True' is stilll redundant.
print(not isFemale) # Much better

# Similar to the above...

# Works, but the is True is redundant.
if isFemale is True:
    print('This person is female')

# Much better
if isFemale:
    print('This person is female')

# Works, but the 'is not True' is redundant.
if isFemale is not True:
    print('This person is not female')

# Works, but the 'is True' is redundant.
if not isFemale is True:
    print('This person is not female')

# Much better
if not isFemale:
    print('This person is not female')
```

Similarly, sometimes an *if* statement is not needed at all:

```
age = 19

# If statement works and is clear, but not necessary.
if age < 18:
    isAdult = False
else:
    isAdult = True

# Same thing, but still not necessary.
isAdult = False if age < 18 else True

# Much better
isAdult = age >= 18

# Same thing, also fine
isAdult = not age < 18
```

5. Every variable type can be converted to a **bool** type. Conceptually, any value that is *worthless* or *empty* will convert to False. Anything else will convert to True. Some examples below:

```
print(bool(True)) # Prints True.
print(bool(False)) # Prints False.

print(bool(None)) # Converts-to and prints False.
print(bool(not None)) # Converts-to and prints True.

print(bool(0)) # Converts-to and prints False.
print(bool(0.0)) # Converts-to and prints False.
print(bool(5)) # Converts-to and prints True.
print(bool(-7)) # Converts-to and prints True.

print(bool([])) # Empty list. Converts-to and prints False.
print(bool([1])) # Non-empty list. Converts-to and prints False.

print(bool(tuple())) # Empty tuple. Converts-to and prints False.
print(bool((1,2))) # Non-empty tuple. Converts-to and prints True.

print(bool({})) # Empty dict. Converts-to and prints False.
print(bool({'Jeff': 25})) # Non-empty tuple dict. Converts-to and prints True.
```

The above may seem like some useless trivia, but it is actually quite relevant: An **if** statement internally converts its expression to a **bool** type. Therefore, the following if statements are valid:

```
# Works, but no need for the 'is not'
if numberOfLoans is not None:
    print('We have some Loans')

# Much better. numberOfLoans converts to a bool implicitly.
# bool(None) is False and bool(and number besides 0) is True.
if numberOfLoans:
    print('We have some Loans')

# Similar, for lists...
people = []

# Works, but inefficient and unnecessary call to len()
if len(people) == 0:
    print('There are no people here')

# Much better
if not people:
    print('There are no people here')

# Now populate the list
people.extend(['Andy', 'Jim', 'Sally'])

# No need for len(people) > 0
if people:
    print('There are people here')
```

TIP: The above syntax is nice and clean; however, it's important to remember that **None** and 0 are not the same thing, though both will evaluate to False. A very common bug in practice (even among experienced developer) is to check if(variable), assuming that the **if** block will only execute if variable is not None. However, if variable contains the number **0**, the **if** block *still* will not execute since **0** is also False. In many situations, this was not

intended and leads to a bug in the code when variable contains **0** (though sometimes this may be intentional, and not a bug).

III: Lists/Tuples

1. Lists, tuples, and other 'iterable' variable types can be accessed with indexing. Python also gives the ability to 'unpack' these variables into multiple variables (one variable per item). See examples below:

   Indexing: Regular way to access individual items of a tuple (or other iterable type).

```
myTuple = (1, 2, 'test', 4)
print(myTuple[0]) # Will print 1
print(myTuple[1])  # Will print 2
print(myTuple[2])  # Will print test
print(myTuple[3])  # Will print 4
```

   Unpacking: This unpacks an iterable type (list, tuple, etc.) into multiple variables.

```
myTuple = (1, 2, 'test', 4)
# v1 will contain 1, v2 will contain 2,
# v3 will contain 'test' and v4 will contain 4.
v1, v2, v3, v4 = myTuple
```

   Discarding Values:

```
myTuple = (1, 2, 'test', 4)
# Let's say we don't care about the first and last values, for whatever reason.
# We can use _ to discard undesired values when unpacking. In this case:
# v1 will contain 2 and v2 will contain 'test'.
_, v1, v2, _ = myTuple
```

   Unpacking with Loops:

```
namesToAges = [('John', 30), ('Sara', 25), ('Gary', 49)]
# Each loop through the outer list will return a tuple.
# Each tuple contains the (name, age)
for tup in namesToAges:
    print(tup[0]) # Prints the name
    print(tup[1]) # Prints the age

# Cleaner approach, using unpacking
# Can choose to ignore unnecessary values with _
# same ad regular unpacking.
for name,age in namesToAges:
    print(name)
    print(age)
```

Unpacking when there is an inner tuple (using loops):

```
namesToAges = [(0, ('John', 30)), (1, ('Sara', 25)), (2, ('Gary', 49))]
# Each loop through the outer list will return a tuple.
# Each tuple contains (index, (name, age))
for tup in namesToAges:
    print(tup[0]) # Prints the index
    print(tup[1][0]) # Prints the name
    print(tup[1][1])  # Prints the age

# Cleaner approach, using semi-unpacking
for i, innerTup in namesToAges:
    print(i) # Prints the index
    print(innerTup[0]) # Prints the name
    print(innerTup[1]) # Prints the age

# Cleanest approach, using unpacking
# Note that the parentheses are necessary due to the inner tuple.
for i,(name,age) in namesToAges:
    print(i)
    print(name)
    print(age)
```

Same as above, using *enumerate*:

```
namesToAges = [('John', 30), ('Sara', 25), ('Gary', 49)]
# Each loop through the outer list will return a tuple.
# Each tuple contains (index, (name, age))
for tup in enumerate(namesToAges):
    print(tup[0]) # Prints the index
    print(tup[1][0]) # Prints the name
    print(tup[1][1])  # Prints the age

# Cleaner approach, using semi-unpacking
for i, innerTup in enumerate(namesToAges):
    print(i) # Prints the index
    print(innerTup[0]) # Prints the name
    print(innerTup[1]) # Prints the age

# Cleanest approach, using unpacking
# Note that the parentheses are necessary due to the inner tuple.
for i,(name,age) in enumerate(namesToAges):
    print(i)
    print(name)
    print(age)
```

2. Lists of the same value can be created as follows. Note the pitfall when creating lists of other lists (or other mutable types):

```
>>> lst = [1]*5
>>> print(lst)
[1, 1, 1, 1, 1]
>>> lst2 = [[]]*5 # A list of five empty lists
>>> lst2[0].append(1) # Will this only append 1 to the first inner list?
>>> print(lst2) # No!! Since the [[]]*5 create five of the same list (similar to the assignment issue)
[[1], [1], [1], [1], [1]]
>>> lst2 = [[] for i in range(5)] # Can do this properly using a list comprehension
>>> print(lst2)
[[], [], [], [], []]
>>> lst2[0].append(1) # This will only affect the first inner list
>>> print(lst2)
[[1], [], [], [], []]
```

Can also create repetitive lists using the same technique (and the same pitfall applies):

```
>>> lst = [1, 6, 5]
>>> lst2 = lst*2
>>> print(lst2)
[1, 6, 5, 1, 6, 5]
>>> print(lst*3)
[1, 6, 5, 1, 6, 5, 1, 6, 5]
>>> lst = [[1, 6, 5]]
>>> lst2 = lst*3
>>> print(lst2)
[[1, 6, 5], [1, 6, 5], [1, 6, 5]]
>>> lst[0][1] = 1000 # This will affect all the inner lists!
>>> print(lst2)
[[1, 1000, 5], [1, 1000, 5], [1, 1000, 5]]
>>> lst2[0][1] = 2000 # This will also affect all the inner lists!
>>> print(lst2)
[[1, 2000, 5], [1, 2000, 5], [1, 2000, 5]]
```

3. Tuples are basically immutable lists (cannot change once created). Tuples can contain one or more values. Syntax to create an empty list, or a list with a single value is simple:

```
l = [] # Empty list
l = list() # Also empty list
l = [1] # List with one value
```

It is also possible to have an empty **tuple** or **tuple** with one value, but the syntax is different than for **list**, in the single-value case:

```
l = () # Empty tuple
l = tuple() # Also empty tuple

# The below is not a tuple,
# since Python thinks (1) is mathematical notation (number 1 in parentheses).
l = (1)
l = (1,) # Proper, single-value tuple
```

4. The same multiplication rules as for **list** apply to **tuple**. The same pitfalls apply as well (since a tuple may contain a list, which is mutable).

5. Functions can only return a single variable. However, this variable may be *any* Python type. This means that returning a tuple is a good way for a function to *appear* to return multiple variables. For example:

```
def divideWithRemainder(numer, denom):
    quotient = numer/denom
    remainder = numer%denom

    # Returns a tuple. Same as return (quotient,remainder),
    # but parentheses not needed to return a tuple, as
    # this helps the illusion of returning multiple values.
    return quotient, remainder

# result gets a tuple.
# This breaks the illusion that divideWithRemainder returned multiple variables.
result = divideWithRemainder(5, 2)
print(type(result)) # Prints <type 'tuple'>
print(result[0]) # Prints the quotient (2)
print(result[1]) # Prints the remainder (1)

# Can use unpacking to complete the multiple return variable illusion
# (see the topic on unpacking tuples)
q, r = divideWithRemainder(5, 2)
print(q) # Prints the quotient (2)
print(r) # Prints the remainder (1)
```

IV: Variables

1. Variables defined outside of any function or class, are called *global variables*. These variables can be accessed from any function within the same script (and may be imported into other scripts). Note that it is common practice to use all caps for global variable names. i.e., GLOBAL_VAR.

2. Multi-assignment is possible. Two examples:

```
# This will assign 12 to x, 15.3 to y, 'test' to z, and 'a' to a.
x, y, z, a = 12, 15.3, 'test', 'a'

# This will assign 15.5 to all the variables.
x = y = z = a = 15.5
```

3. It's important to understand the concept of variable *scope*. Variable scope is essentially where a given variable exists (where it can be accessed from):
    a. A variable defined within a function is only accessible from within that function.
    b. The above is also true for class functions (see next level).
    c. A variable defined within a class (but outside any function) is only accessible from within or on that class, or an object of that class (see next level).
    d. A *global* variable is one that is defined in a script, but outside any function or class. This variable is accessible (to *read*) from any place or function within the same script. It is also accessible to *write* from any place or function within the same script, but this requires the **global** keyword:

```
# Global variable (note all caps).
MY_VALUE = 10

def myFunction():
    print(MY_VALUE)

def myFunctionIncorrectGlobalWrite():
    # This does *not* modify the global variable.
    # Rather, it creates a new locally-scoped variable with the same name.
    # This locally-scoped variable is only accessible within this function.
    MY_VALUE = 20
    print(MY_VALUE) # Prints 20

def myFunctionGlobalWrite():
    # This tells Python that MY_VALUE within this
    # function is meant to be the global one.
    global MY_VALUE
    MY_VALUE = 20 # Modifies the global variable.
    print(MY_VALUE) # Prints 20

myFunction() # Prints 10
myFunctionIncorrectGlobalWrite() # Prints 20
print(MY_VALUE) # Prints 10 since the global MY_VALUE was not changed.

myFunctionGlobalWrite() # Prints 29
print(MY_VALUE) # Prints 20!
```

Global variables are not accessible outside the script they're defined in, unless they're imported. You can import a global variable explicitly or using the full module path, the same as you do for importing functions.

Variables within a given scope must have a unique name. However, two variables may share a name if they are defined within different scopes.

4. When assigning one variable is another in Python, both variables will actually refer to the same value (Python does not copy it). For simple (non-mutable) types (i.e., **int**, **float**, **str**, **tuple**) this does not really affect you. However, for mutable types (i.e., **list**, **dict**, user-defined types) this is a big deal. This was discussed in the lecture, but see below for some more examples on how it can affect you:

```
>>> l1 = [1, 2, 3]
>>> l2 = l1
>>> print(l1)
[1, 2, 3]
>>> print(l2)
[1, 2, 3]
>>> l1.append(5)
>>> print(l1)
[1, 2, 3, 5]
>>> print(l2) # l2 changed as well!
[1, 2, 3, 5]
>>> print(l1 == l2)
True
>>> print(l1 is l2) # They are the same!
True
>>> l2 = [1, 2, 3] # This 'breaks the connection' since l2 is now its own list
>>> print(l2)
[1, 2, 3]
>>> print(l1)
[1, 2, 3, 5]
```

To avoid this (if necessary), can force Python to copy the variables:

```
>>> l1 = [1, 2, 3]
>>> import copy
>>> l2 = copy.copy(l1) # Copies l1
>>> print(l1)
[1, 2, 3]
>>> print(l2)
[1, 2, 3]
>>> l1.append(5)
>>> print(l1)
[1, 2, 3, 5]
>>> print(l2) # l2 ot affected since it was copied
[1, 2, 3]
```

However, copy is not adequate for lists that contain other lists (or any mutable type that contains another mutable type):

```
>>> l1 = [1, 2, [1, 2, 3]]
>>> import copy
>>> l2 = copy.copy(l1)
>>> print(l1)
[1, 2, [1, 2, 3]]
>>> print(l2)
[1, 2, [1, 2, 3]]
>>> l1.append(5) # This will not affect l2
>>> print(l1)
[1, 2, [1, 2, 3], 5]
>>> print(l2)
[1, 2, [1, 2, 3]]
>>> l1[2].append(7) # This WILL affect l2, since ;1[2] is a list as well, and this inner list was NOT copied
>>> print(l1)
[1, 2, [1, 2, 3, 7], 5]
>>> print(l2)
[1, 2, [1, 2, 3, 7]]
```

To avoid this (when necessary), perform a deepcopy (which will copy any number of layers deep):

```
>>> l1 = [1, 2, [1, 2, 3]]
>>> import copy
>>> l2 = copy.deepcopy(l1)
>>> print(l1)
[1, 2, [1, 2, 3]]
>>> print(l2)
[1, 2, [1, 2, 3]]
>>> l1[2].append(7) # This will no longer affect l2, since the inner list was copied.
>>> print(l1)
[1, 2, [1, 2, 3, 7]]
>>> print(l2)
[1, 2, [1, 2, 3]]
```

Note that the same thing applies to passing mutable variables as arguments to a function: The function parameter variable references the same value as the calling function, so any changes to it *will* affect the variable for the calling function as well (unless you copy it).

V: Functions

1. We've seen functions that return a value. The type of value returned may be anything from a number, string, tuple, list, dictionary, or any other Python type. The return value may even be **None** (which is NoneType). The default return value of a functions is None. Therefore, when returning **None**, there is no need to explicitly do so. A couple examples below:

```
def monthlyRate(annualRate):
    if type(annualRate) is float and annualRate >= 0.0:
        return annualRate/12.0
    else:
        # This is not necessary. See next example.
        return None

print(monthlyRate(3.12)) # Will print 0.26
print(monthlyRate(-5.0)) # Will print None
```

This is the same as above, without the explicit returning of None:

```
def monthlyRate(annualRate):
    if type(annualRate) is float and annualRate >= 0.0:
        return annualRate/12.0

print(monthlyRate(3.12)) # Will print 0.26
print(monthlyRate(-5.0)) # Will print None
```

2. Sometimes it becomes necessary to add a new parameter to a function, in addition to the existing parameters. When dealing with a large code base (i.e., in a bank), it's important to always add the new parameter at the end (on the right side) of the parameter list, and not somewhere in the middle. Additionally, the new parameter should always have a default value (optional). This ensures backwards-compatibility of the code, for functions that are reliant on the parameter order as-is. Example:

Original function:

```
def MyFunction(var1, var2):
    print var1, var2

MyFunction(21, var2=6)
```

Incorrect Approach - Add var0 on the left; this causes the **MyFunction** call to break for two reasons: The function now requires another non-keyword argument, and *21* is now being passed-into **var0**, when it was originally intended to be passed-into **var1**:

```
def MyFunction(var0, var1, var2):
    print(var0, var1, var2)

MyFunction(21, var2=6)
```

Correct Approach - Add var0 on the right, and make it optional:

```
def MyFunction(var1, var2, var0=None):
    print(var0, var1, var2)

MyFunction(21, var2=6)
```

3. It is possible to forward *args and **kwargs parameters from one function to another. Additionally, it is possible to unpack a tuple or list to fill the parameters of a function. We can do this using * and ** on a tuple/list or dict respectively. Simply passing *args* or *kwargs* without * or ** results in a single variable (the actual tuple, list, or dict) getting passed into the function as a single argument, instead of it being expanded into the function's parameter list. A few examples to illustrate below:

Parameter Forwarding (unpacks *args and **kwargs to reuse as parameters.

```
def myDelegationFunc(a, b, *args, **kwargs):
    print(a,b)
    print(args) # args is a tuple
    print(kwargs) # kwargs is a dict

def myFunc(a, b, *args, **kwargs):
    # To forward *args and **kwargs to another function, simply pass them in using * and ** syntax.
    myDelegationFunc(a, b, *args, **kwargs)

myFunc(1, 2, 3, 4, 5, kwd=15, kwd2=17)
```

The below leverages * to unpack a list into the multiple parameters of zip:

```
>>> players = ['James', 'Melvin', 'Bruno']
>>> weights = [225, 190, 213]
>>> playersToWeights = zip(players, weights)
>>> print(playersToWeights) # zip object contains [('James', 225), ('Melvin', 190), ('Bruno', 213)]
<zip object at 0x03942B08>
>>> reverseZip = zip(*playersToWeights)
>>> print(reverseZip) # zip object contains [('James', 'Melvin', 'Bruno'), (225, 190, 213)]
<zip object at 0x039586C8>
>>> reverseZipAgain = zip(*reverseZip)
>>> print(reverseZipAgain) # zip object contains [('James', 225), ('Melvin', 190), ('Bruno', 213)]
<zip object at 0x03958BA8>
```

VI: Built-in Functions

1. It is possible to round numbers using the **round** function. Additionally, one can truncate the decimal off a float value by using the **int** function. For example:

```
val = 1.55785
roundedVal = round(val, 2) # Will round to 1.56
roundedVal2 = round(val, 0) # Will round to 2.0
flooredVal = int(roundedVal2) # Will remove the .0 decimal
roundedVal3 = round(val, 1) # Will round to 1.6
```

It's also important to note that decimal values may look somewhat counterintuitive. For example, roundedVal3 above will actually contain `1.6000000000000001`. This is because of internal machine error when dealing with decimals (beyond the scope of this course). Due to this, when comparing decimal values (i.e., when using an *if* statement), it's a good idea to compare within a tolerance instead of comparing strict equivalence. For example:

```
val1 = 1.60000000001
val2 = 1.6

# Check for strict equivalency.
# This may not be desirable, due to machine precision errors.
if val1 == val2:
    print('Equal!')

# Compare within tolerance.
# Can specify any tolerance that makes sense in your context.
tol = .00000001
if abs(val1-val2) < tol:
    print('Equal!')
```

VIII: Loops

1. We've seen how to use a **for** loop in Python. One interesting feature in Python is combining the **else** statement with a **for** loop (we've only seen **else** in the context of **if** statements until now). This can be very handy in certain situations: Essentially, the **else** clause of a for loop will execute if the loop either did not **break** early or if the loop did not iterate at all. The following two examples illustrates this:

```
enteredList = input('Enter a list of values, separated by commas: ')

for value in enteredList.split(','):
    if float(value) == 100:
        print('\nNumber 100 was found!')
        break
else:
    print('\nThe list was either empty or 100 was not found.')
```

The problem with the above approach is that there is no way to differentiate, within the **else** clause, whether it's executing due to the list being empty or because the **break** statement never hit. We can resolve this with the following tweak:

```
enteredList = input('Enter a list of values, separated by commas: ')
loopStarted = False
for value in enteredList.split(','):
    loopStarted = True
    if float(value) == 100:
        print('\nNumber 100 was found!')
        break
else:
    if loopStarted:
        print('\nThe list was empty')
    else:
        print('\n100 was not found.')
```

Note that **else** also works for **while** loops, but the behavior differs somewhat from the **for** loop:

```
enteredList = input('Enter a list of values, separated by commas: ').split(',')
while enteredList:
    # Pop removes and returns the first value from the list.
    value = enteredList.pop()
    if float(value) == 100:
        print('\nNumber 100 was found!')
        break
else:
    print('\nThis will execute in every case 100 is not encountered.')
```

IX: Console Output

1. We've primarily been using the **print** statement for console output. The **print** statement is limited when it comes to displaying complex types such as lists or dicts. The **pprint** statement can be really handy for elegantly displaying a large dict or list. See example below for **dict**:

```
>>> import pprint
>>> myDict = {'Greg':25, 'Sally':35, 'Jennifer':45, 'Paul':21, 'Tom':65, 'Teddy':12, 'Cal':39, 'Sarah':41}
>>> print(myDict)
{'Greg': 25, 'Sally': 35, 'Jennifer': 45, 'Paul': 21, 'Tom': 65, 'Teddy': 12, 'Cal': 39, 'Sarah': 41}
>>> pprint.pprint(myDict)
{'Cal': 39,
 'Greg': 25,
 'Jennifer': 45,
 'Paul': 21,
 'Sally': 35,
 'Sarah': 41,
 'Teddy': 12,
 'Tom': 65}
```

It's even more useful for a dict of dicts:

```
>>> import pprint
>>> myDict = {'Greg': {'age':25, 'height':69}, 'Sally':{'age':35, 'height':60}, 'Jennifer':{'age':19, 'height':52}}
>>> pprint.pprint(myDict)
{'Greg': {'age': 25, 'height': 69},
 'Jennifer': {'age': 19, 'height': 52},
 'Sally': {'age': 35, 'height': 60}}
```

X: Packages, Modules, and Imports

1. We discussed importing built-in and custom modules/packages. One thing not mentioned was best practices is regards to importing. Some points below:
    a. Explicit importing of names (**from…import**) is generally preferred when importing individual classes, functions, or variables from a module that contains many. Full-path module imports (**import a.b.c**) is generally better if a module contains many functions, classes, or variables that need to be used. For module import with large paths, you can **import a.b.c.d.e.f.g as g** to avoid having to explicitly specify the entire module path everywhere it is used; however, the **as** should always be identical to the last name in the module path (**g** in this example).
    b. When using explicit imports, it's good to be aware of 'circular import' issues. For example:

```
'''
script1.py
'''
from script2 import function3

# This function relies on function3
def function1():
    return function3()/2
'''
script2.py
'''
from script1 import function1

# This function relies on function1
def function2():
    print(function1()*5)

def function3():
    return 10
'''
main.py
'''
from script2 import function2 # ImportError: cannot import name function3

function2()
```

The above error occurs since script2 relies on function1 from script1 and scrip1 relies on function3 from script2. There are two approaches to resolving circular imports. One approach is importing on-demand (see below bullet-point). The second approach is using full module imports:

```
'''
script1.py
'''
import script2

# This function relies on function3
def function1():
    return script2.function3() / 2
'''
script2.py
'''
import script1

# This function relies on function1
def function2():
    print(script1.function1() * 5)

def function3():
    return 10


'''
main.py
'''
from script2 import function2 # Works now!

function2()
```

c. When importing modules, all global code gets executed during import. This includes the module's own imports, any global variables, and any function calls. Example below:

```
'''
script1.py
'''
import script2 # This gets executed as soon as this script is imported anywhere.
myGlobalVariable = 10 # This gets executed as soon as this script is imported anywhere.
script2.function4() # This gets executed as soon as this script is imported anywhere.

# This function gets created, but does not execute on import.
# It only executes if the function is called.
def function1():
    return script2.function3()/2

'''
script2.py
'''
import script1

# This function relies on function1.
def function2():
    print(script1.function1()*5)

def function3():
    return 10

def function4():
    print('In function 4')

'''
main.py
'''
# This imports script1 and executes all its global code.
# Since script1 calls script2.function() globally,
# this import causes 'In function 4' to print.
import script1

# myGlobalVariable was also created/initialized on import of script1.
print(script1.myGlobalVariable) # Prints 10
```

d. Imports should almost always be at the top of the script (after the doc-string). In general, import all your own modules first then import built-in modules.

e. It's *possible* to import anywhere in your code (at the top, within a function, or within a class). There are two possible benefits of importing within a function or class (though it should be avoided whenever possible):

- It is one possible way to avoid the circular import issue.
- If a specific module takes a long time to import (i.e., if it does some heavy processing on import), you may wish to avoid the import until necessary (i.e., only when the function that uses the heavy module executes).

Example below:

```
'''
script1.py
'''
# This function gets created, but does not execute on import.
# It only executes if the function is called.
def function1():
    from script2 import function3 # This import only happens when function1 is called.
    return function3()/2

'''
script2.py
'''
import script1

# This function relies on function1.
def function2():
    from script1 import function1 # This import only happens when function2 is called.
    print(function1()*5)

def function3():
    return 10

def function4():
    print('In function 4')

'''
main.py
'''
from script2 import function2 # Works!

function2() # Prints 25
```

2. We've been using the **__name__ == '__main__'** syntax throughout our code. Now that we have discussed the details of importing, here is a brief explanation of what this does and how it works: As mentioned above, when you import a script, all global functionality automatically executes. However, functions do not execute unless explicitly invoked. The same thing applies to executing a script (i.e., by clicking Run in PyCharm) – all global code in that script will execute but functions will not. The **main** function is no different than any other function – it will not execute unless it is explicitly invoked.

The purpose of the **main** function is to define an *entry point* into your program – **main** is expected to execute first and contains all code and function calls necessary to run the program. Therefore, we wish for the **main** function to immediately run when executing a script that contains a **main** function. The caveat here is that if you *import* a script which has a **main** function, you do *not* want that **main** function to run.

To achieve this behavior of only running **main** when executing a script but not when importing a script, we first need a way to identify whether or not a script was executed or imported. This can be done by checking the

contents of the built-in global **__name__** variable: If it contains '__main__', then you know that the script was executed directly. Otherwise, the script was imported. Therefore, the following code achieves our goal:

```
# This is global code that executes when the module is imported or executed.
# If the script was executed directly, __name__ will contain '__main__'.
if __name__ == '__main__':
    # Since the script was executed directly, we invoke the main() function.
    main()
```

3. Can import everything from a module without having to explicitly name each function. The syntax is **from…import *.** However, you should never use this in practice (memory considerations among others).