Lecture 7

Intro to Programmer-Defined Functions; Local Variables; Lists and Loops; Lists in Detail; Mutation Operations; What?; Calculator

1. A Quick Introduction to Programmer-Defined Functions

We have used a lot of functions in Python: `len()`, `print()`, `math.exp()`, `random.randrange()`, etc. As a programmer, you can write your own functions, too!

Why would you want to? Because **functions can help you break down your problems into smaller, more manageable parts** and **minimize repetition of code**. The act of writing your programs in pieces is sometimes called *modular programming*.

Before justifying that this really is useful, let's look at an example, which illustrates the mechanics (if not the purpose) of programmer-defined functions.

In [ ]:
```
# EXAMPLE 1a: A programmer-defined function
# This function computes sums of squares of numbers.
# Presumably, this is something that we may find ourselves doing a lot.

def sum_of_squares(x, y, z):
    sos = x**2 + y**2 + z**2
    return sos
```

Run that code. It doesn't appear to really do much! But it defines a new function. The keyword `def` denotes that we're defining a function; `sum_of_squares` is the *name* of the function; `x`, `y` and `z` represent the *inputs* to the function; and the word `return` denotes that the *output* of the function is the value called `sos`.

Now, let's **use** the function.

In [ ]:
```
# EXAMPLE 1b: Using that function

def sum_of_squares(x, y, z):
    sos = x**2 + y**2 + z**2
    return sos

##################################
# Here's where the action starts.

# On this line, I am CALLING (using) the function.
# The variables x, y and z temporarily become the inputs 1, 3, 5, respectively.
# When you USE the function, you don't write "def"
# (the same way that you haven't written def when you use len() or math.sqrt())
a = sum_of_squares(1,3,5)
print(a)

# A function is meant to be reused!  I don't rewrite the definition of the function
,
# but I use it multiple times.
b = sum_of_squares(1,4,0)
print(b)

# Of course, these three lines happen to do nothing, for exactly the same reason...
3 + 5                  #
len("Hello")           # They all compute values, which then go unused.
sum_of_squares(1,1,1)  #


# You can use variables and expressions as inputs, as well.
p = 4
q = 1
print(sum_of_squares(p, 2*q, p + q))
```

Let's go through this. There are two things to understand -- how the function is *defined*, and how it is used (or *called*). Let's start with the definition.

In [ ]:
```
BASIC FUNCTION DEFINITION SYNTAX:

def <function name>(<parameter list>):
    <body>
    return <output value>
```

The word `parameters` refers to names used for the inputs when you define the function. So in the program above, the parameters were `x`, `y` and `z`, and the output value was called `sos`.

When your program encounters a function definition, it does .... well, not much, at first. This is just a *definition* of a function -- what you see above the ################## in this example.

But then, later, you may *call* this function, which would look something like

`{function name}(<actual parameters>)`

Calls to a function must occur lower in the code than where that function is defined. In our program, the calls looked like `sum_of_squares(var1, var2, var3)`. When Python encounters `sum_of_squares(var1, var2, var3)`, program control is passed to the function -- that is, the function will stop its normal execution, and instead executing the function's code. More specifically, here is what happens:

- The parameters of the function will be matched with the actual parameters (or *arguments*). This means that `x` will be assigned the value of `var1`, `y` will be assigned the value of `var2`, and `z` will be assigned the value of `var3`. (First gets matched with first, second gets matched with second -- order matters here.)
- The body of the function will execute, using these values.
- The line `return sos` ends the function's execution, and the value of `sum_of_squares(var1, var2, var3)` will be whatever `sos` is.
- That value then gets stored to a variable, or printed, or ignored, depending on what you code does with `sum_of_squares(var1, var2, var3)`, and the program continues.

Let's write one together. (This would be a very useful function to write, were it not actually already part of the Python language.)

In [ ]:
```
# EXAMPLE 1c: Max function
# Let's write a function called my_max.
# This function should receive two arguments, are output whichever is greater.

def my_max(first, sec):
    if first > sec:
        output = first
    else:
        output = sec
    return output

a = 3
b = 4
c = 8

print("my_max(a,b) =", my_max(a, b))
print("my_max(c+1,b) =", my_max(c + 1, b))
print("my_max(5,c) =", my_max(5, c))
```

Try writing a function called `my_abs`, which should receive one numerical argument, and return the absolute value of that argument.

In [ ]:
```
# EXAMPLE 1d: my_abs

def my_abs(x):      # ALTERNATIVE BODY:
    if x > 0:       # if x > 0:
        ab = x      #     return x
    else:           # else:
        ab = -x     #     return -x
    return ab

print("If this isn't 2, your function has a problem :", my_abs(2))
print("If this isn't 4, your function has a problem :", my_abs(-4))
print("If this isn't 0, your function has a problem :", my_abs(0))
```

2. Local Variables in Functions

Variables that are defined in a function are **local** variables: these are variables that will only exist when a function is being executed, and cease to exist when the function is finished executing. So, for example, if you define a variable named `xyz` inside the body of a function, then you can only refer to that variable within the function. If you then also define a variable with the same name `xyz` outside of the function, then -- oddly enough -- Python will treat those as different variables (for reasons that we'll come back to later).

This means that: you can't modify outside variables inside a function (except for assigning a return value); and you can't access local variables outside of the function they are defined in. This can be a little confusing, but fortunately there are two general principles for writing functions which usually will help you build working, helpful functions:

1. The only outside values that a function uses should be the *arguments*.
2. The only effects that a function should have on program execution should be through its *returned value*.

I'll call these the "One Way In" and "One Way Out" principles.

Everything in the preceding two paragraphs is an oversimplification, and we'll elaborate on it down the road. For now, you should follow this advice for now if you write functions: if you don't, there's a good chance that your functions will end up not working or being confusing and unhelpful.

For now, look at the following examples, of things **not** to do.

In [ ]:
```
# EXAMPLE 2a: Local variables, part I

x = 20


# We'll try to write a function to update the value of x.
# It won't work, because the x inside in the function is a local variable.
def x_times_y(y):
    x = x*y     # On the right side, where does the value of x come from?
                # It doesn't come from an argument.
                # So I guess it comes from the x = 20 line above?
                # But then we're violating the One Way In principle.
                # (And, not coincidentally, we get an error)
    return x

print(x_times_y(10))
```

In [ ]:
```
# EXAMPLE 2b: Local variables, part II


def fn(z):
    new_guy = z + 1
    z = z + 2
    return z


x = fn(40)

# We're trying to use a value that was created by fn.  This violates the
# One Way Out principle -- the only effect that fn should have on program execution
# should be through a return value.  (And not coincidentally, we get an error.)
print(new_guy)


# This violates the One Way Out principle, too.  Don't be confused by the fact that
we
# "return z":  it's not really the NAME z that is returned by the function, it's th
e VALUE
# held by z that is returned (and in this case, stored to x).  References to z outs
ide of
# the function body will also cause errors.
print(z)
```

In [ ]:
```
# EXAMPLE 2c: Don't print, return!

# One Way Out principle: the only effect that fn should have on program execution
# should be through a RETURN value.  Functions (usually) shouldn't PRINT: they shou
ld
# RETURN!

def bad_max(x, y):
    if x > y:
        output = x
    else:
        output = y
    print(output)  # NOOOOOO :( :(


# At first this seems fine, mostly. (Although you get these weird "None"'s.)
print(bad_max(3,7))
print(bad_max(5,-1))

# But what if you want to do something fancier?
x = bad_max(1,2) + bad_max(5,3)
print(x)
```

3. Lists and Loops

Suppose I want to print out all the months of the year, like so:

```
Month number 1 is January
Month number 2 is February
Month number 3 is March
Month number 4 is April
Month number 5 is May
Month number 6 is June
Month number 7 is July
Month number 8 is August
Month number 9 is September
Month number 10 is October
Month number 11 is November
Month number 12 is December
```

There's a lot of repetition. We could cut and paste `print("Month number")` over and over again, but perhaps there is a better way. Like.....this.

```
In [ ]: # EXAMPLE 3a: The months of the year

        months = ["January", "February", "March", "April", "May", "June", "July", "August",
        "September", "October", "November", "December"]

        num = 1

        for m in months:
            print("Month number " + str(num) + " is " + m)
            num = num + 1
```

We've just introduced two things at once:

- A new data type -- the ***list***
- and a new type of statement -- the ***for loop***

Hey everyone! This is it! This is where we start programming for real! Get ready!

All of the programs we've written until now can only execute as many commands as you've written. If your program has 20 lines, the computer will execute 20 (or fewer) instructions. That's fine, but the power of having a computer at your disposal is its *speed*. We want it to be able to execute millions of instructions, but we don't want to have to *write* millions of lines, or *manually enter* millions of pieces of data. With lists and loops, we can now use minimal code to execute huge numbers of instructions.

One more example: what is the sum of the first hundred million positive integers?

```
In [ ]: # EXAMPLE 3b: 1+2+3+4+....+100,000,000

        running_sum = 0

        for n in range(1, 10 ** 8 + 1):
            running_sum = running_sum + n

        print(running_sum)
```

The computer just did 100 million operations while you watched! (Actually, way more than 100 million operations, since Python has to do many things in the background to perform each addition.)

4. Lists in Detail

A ***list*** is a collection of values, arranged in an order. The basic way to create a list is with the following syntax:

```
In [ ]: LIST SYNTAX:

        <list name> = [<first value>, <second value>, <...and so on...>, <last value>]
```

Note that a list is enclosed in square brackets `[` and `]`. If you use regular parentheses, you're creating something else that may end up working for your purposes, but it is definitely not the same type of list that I'm talking about.

The values can be of any types, and don't all have to be the same type.

```
In [ ]: #EXAMPLE 4a: Lists

        x = [12, "Hello", 13, "Goodbye", True, 14]
        y = [5, 6, 7]

        print("type(x) =", type(x))
        print("x[2] =", x[2])
        print("x[-1] =", x[-1])
        print("x[-2] =", x[-2])
        print("len(x) =", len(x))
        print("x + y =", x + y)
        print("y*3 =", y*3)
        print("x[626] =", x[626])
```

As you can tell, we just illustrated several features of lists.

- First, `list` is a new data type.
- Remember how you can index the characters of a `str` using `[ ]`? You can do the same thing with `lists`. And just like with `strs`, **zero-based indexing** is employed.
- You can also do indexing based from the end: e.g., `x[-1]` is the last element of `x`, `x[-2]` is the second-to-last, etc.
- If you try to read an index that is longer than the length of the list, however, you will get a run-time error.
- `len()` also works for lists.
- You can concatenate `lists` using `+`: this will produce a new `list`, which combines the two operands.
- And you can multiply a `list` by an integer.

I should probably also show you ***slicing*** while we're covering the basics. Slicing refers to taking a portion of a list (not necessarily just a single element). To do this, you would write an expression that looks like, e.g., `x[2:6]`, which would create a new list, containing elements `x[2]`,`x[3]`, `x[4]`, `x[5]` -- but not `x[6]`. (That took me a while to get used to -- the last index is *not* included, just the ones before.) Let's look at examples below.

In [ ]:
```
#EXAMPLE 4b: Slicing

my_list = ["Exhibit A","Exhibit B","Exhibit C","Exhibit D","Exhibit E","Exhibit F",
"Exhibit G","Exhibit H"]

# A basic slice; this contains elements 3,4,5, and 6 (using 0-based indexing)
slice_1 = my_list[3:7]
print("slice_1 =", slice_1)

# If you leave out the part before the :, it assumes you're starting at the beginni
ng;
# if you leave out the part after the :, it assumes you're going until the end.
print("my_list[:2] =", my_list[:2])
print("my_list[6:] =", my_list[6:])

# You can even use negative indices in slices.
print("my_list[-6:4] =", my_list[-6:4])

# What happens if you try to slice out of bounds?
print("my_list[100:102] = ", my_list[100:102])

# You can also slice strings, by the way.
big_string = "abcdefghi"
print(big_string[2:5])
```

5. Lists, Mutability and The Basic Mutation Operations

You may have notice that `strs` and `lists` are a lot alike: indeed, a `str` is pretty much -- not quite, but pretty much -- a list where each element must be a single character. One big difference is that `strs` are **immutable**, whereas `lists` are **mutable**.

Let me show you the major implications of this, before returning to describe exactly what this means.

In [ ]:
```
# EXAMPLE 5a: Strings and lists
# List objects can change, while string objects cannot.

my_list = ["I", "can", "change"]
my_string = "I canNOT change"

# Here's the basic difference: you can change parts of the list.
my_list[2] = "changggggggeeee!"
my_list.append("See? Now I just got longer too")
print(my_list)


# But the following analogous operations with strings don't work.
my_string[2] = "B"
my_string.append("!!!")
print(my_string)

# Note that this does NOT mean that my_string can't change.  It's just that
# if you change a variable, you have to change the entire variable.
my_string = "If I change the whole string, it's ok"
my_string = my_string + " (and this line is ok too)"

my_list = ["I", "can also", "change entire lists", "of course"]

print(my_string)
print(my_list)
```

So, you change **parts** of `lists`, but if you want to change a `str`, you have to create an entirely *new* `str`. Ok, this isn't exactly the full reasoning; we'll get to that later.

Let's talk about some "mutation" operations for lists.

- `.append({elt})` puts *{elt}* at the end of a list.
- `.insert({pos}, {elt})` puts *{elt}* into the list at position *{pos}* (with the element currently at that position moving back).
- `del {list name}[{pos}]` removes element *{pos}* from the list *{list name}*.

Other useful ones that I encourage you to look up include `.extend()`, `.remove()`, and `.pop()`.

In [ ]:
```
# EXAMPLE 5b: List operations

x = ["a", "b", "c", "d", "e"]
# Add f and g
x.append("f")
x.insert(3,"g")
print(x)

# Let's take them back out using del
del x[3]
del x[5]
print(x)
```

6. Pause: What?

In [ ]:
```
# EXAMPLE 6a: Assignment

# Compare the results of this bit of code....
x = "Old"
y = x
x = "New"
print(y)

# ...with this one...
x = ["Old", "One"]
y = x
x = ["New", "Guy"]
print(y)

#...AND THIS ONE!!!!!
x = ["Old", "One"]
y = x
x[0] = "New!!!!"
print(y)
```

Why are the last two snippets so different? They both change `x`, but one change causes `y` to change, and the other doesn't.

The short short short answer is: be careful when you copy one list variable into another variable, especially when you perform mutation operations. The same goes for any mutable data types -- for now, lists are basically the only ones we know about.

The long answer will come at a later date. It will either be really enlightening or it will shatter your youthful optimism once and for all.

7. Calculator

Let's write a program that reads a line like

```
12.3 - 3.4
```

from user input, interprets it as an arithmetic expression, and the evaluates that expression. Note that this is **not** as simple as it sounds, for one basic reason:

user input is always interpreted as a `str`, not as code!

As far as Python is concerned, when you enter `2 + 3`, it just sees a 5-character `str`: a 2 symbol, a space, a + symbol, a space, and a 3 symbol. Of course, the `int` and `float` functions can help interpret `strs` as numbers if each character happens to be a digit or a decimal mark. But beyond that, there aren't many standard solutions for converting strings into code. (This actually is a lie -- there is a function that convertes input to executable code, but it is a **huge** security risk, whose use in serious code should be considered a last resort, and so we will avoid it.)

So, how do we do this?

- Get input, as a `str`.
- Second, break apart that `str` into a number, an operation symbol, and another number.
- Check if the operation symbol is `"+"`, `"-"`, etc.
- Once we have determined the operation symbol, evaluate a Python expression with the corresponding operation, and print the answer.

To perform the second step, let me introduce a really useful function: `.split()`. If `str_var` is a `str` variable, then `str_var.split()` is an expression which, when evaluated, produces a list, whose entries are the consecutive runs of non-whitespace characters in `str_var`. ("Whitespace" refers to spaces, tabs, and newlines.) That's a mouthful, but when you see it laid out it isn't so bad. For example:

In [ ]:
```
# EXAMPLE 7a: .split()

a = "    Hey   Ho     Let's      Go   !"
print(a.split())

b = "Got\tsome\ttabs"
print(b.split())

c = """
Big
Multi
Line
"""
print(c.split())
```

So, our program will accept a long `str`, presumably with spaces before and after the operation symbol. Assuming this, we can take the input and use `.split()` to break it into 3 pieces. Let's see the final code.

In [ ]:
```
# EXAMPLE 7b: Calculator

expression = input("Enter an expression: ")

symbols = expression.split()

argument1 = float(symbols[0])
argument2 = float(symbols[2])
operation = symbols[1]

if operation == "+":
    print("=", argument1 + argument2)
elif operation == "-":
    print("=", argument1 - argument2)
elif operation == "*":
    print("=", argument1 * argument2)
elif operation == "/":
    if argument2 == 0:
        print("Can't divide by 0 silly")
    else:
        print("=", argument1 / argument2)
else:
    print("Unparsable expression")
```