CptS 111 — Lab #6

More Practice with Functions

Learning Objectives:

- Use `pass` as function stub
- Write functions with lists as parameters
- Define more complicated functions
- Use of `main()`
- Use of the `global` command
- Open and write to a file

Prerequisites:

- Know list basics and conditionals
- Understand both void and non-void functions

Task 1: Using the IDLE Shell window to practice functions

We'll practice a few new concepts related to functions using the IDLE Shell window. Again, type everything given below in boldface, and remember that it's important for you to understand the comments!

```
>>> # When we're writing complex programs with many functions, it's
>>> # useful to develop a ``skeleton'' of the program and complete and
>>> # test one function at a time.  We can do this using function stubs.
>>> def det_anchor():
        pass
>>> def calc_doom():
        pass
>>> def det_day():
        pass
>>> det_anchor()
>>> calc_doom()
>>> det_day()
>>>
>>> # We defined three function names, but we used pass to indicate
>>> # the function should be ignored.  Then we called the functions
>>> # without throwing exceptions, i.e., causing errors.
>>>
>>> # Then we can define one function and test it to make sure it works
>>> # before writing the next as shown below.
>>>
>>> def det_anchor(num):
        if num == 19:
            return 3
        elif num == 20:
            return 2
        else:
            print('Invalid date used.')
        return

>>> anchor = det_anchor(20)
>>> anchor
2
>>> # After we know one function is correct, we can move on to the next.
>>> # However, we won't do this now.  Instead, let's examine the use of
>>> # lists with functions.  Consider the VOID function below.
>>> def add_name(nlist):
        name = input('Enter the name you want to add: ')
        nlist.append(name)
        return
>>>
>>> # Next, define the list names and call add_names.
>>> names = ['Sam', 'Mohammed', 'Ann', 'Maria']
>>> add_name(names)
Enter the name you want to add: Jin
>>> names
['Sam', 'Mohammed', 'Ann', 'Maria', 'Jin']
>>>
>>> # What has happened here?  We defined the function add_name(),
>>> # created a list, and passed this list to the function.  We didn't
>>> # return the updated list, and yet 'names' includes the name we
>>> # entered.  This is because lists are mutable!  We passed the list
>>> # to the function, and even though it was given another name, it
>>> # was still the same list.
>>> def add_name(ntup):
        names = list(ntup)
        name = input('Enter the name you want to add: ')
        names.append(name)
        ntup = tuple(names)
        return
>>>
>>> names = ('Ann', 'Sam', 'Mohammed', 'Maria')
>>> add_name(names)
Enter the name you want to add: Jin
>>> names
('Ann', 'Sam', 'Mohammed', 'Maria')
>>>
>>> # In the example above, we used a tuple which we converted to a
>>> # list so we could add the name.  Then we converted it back, but
>>> # because tuples aren't mutable, the original tuple didn't
>>> # change.  However, if we returned the updated tuple, it would
>>> # include the added name. ***PROVE THIS ON YOUR OWN!*** Remember
>>> # to change add_name to a non-void function and to assign an lvalue
>>> # to the function call.
>>>
>>> # Let’s turn to scope now.  All lvalues created in a function
>>> # are LOCAL to the function.  Anything outside the function is
>>> # GLOBAL and can be found by the function.
>>> i = 0
>>> def inc():
        i += 1
        return i
>>> inc()
>>>
>>> # Wait!  What happened?  Why didn’t the function find ’i’ when
>>> # we initialized it globally, i.e., outside the function?  IT’S
>>> # BECAUSE LVALUES IN A FUNCTION ARE ALWAYS LOCAL TO A FUNCTION;
>>> # THAT’S WHERE THEY’RE DEFINED!  Recall that function parameters
>>> # are also local to the function.
>>> num1 = 5
>>> def add_10():
        num = num1 + 10
        return num
>>> num2 = add_10()
>>> num2
15
>>> # This time we had no problem finding ’num1’ because it was
>>> # defined globally and wasn’t used as an lvalue.  We can
>>> # actually use a command that will allow a function to see a
>>> # global variable and also use it as an lvalue.
>>>
>>> i = 1
>>> def inc():
        global i
        i += 1
        return
>>> inc()
>>> i
>>>
>>> # As we see, we simply have to use the global command, and the
>>> # function is able to use ’i’ as an lvalue.  Also, we didn’t
>>> # return ’i’ and yet its new value is available globally.  Use
>>> # of the global command is required in PA #4.
>>> #
>>> # Next we’re going to change directions and learn how to open a
>>> # file and print to it.  This is required for PA #4 where it’s
>>> # explained, but let’s consider it here.
>>> filnam = ’my_info.txt’
>>> file_out = open(filnam, ’w’)
>>> print(’My name is SpongeBob.’, file=file_out)
>>> file_out.close()
```

```
>>>
>>> # In the example above, filnam is a string, and we use 'w' to
>>> # write to the file.  However, using 'w' can be dangerous because
>>> # if a file exists with the name 'my_info.txt', all its contents
>>> # will be erased.  Note that we also need to close the file using
>>> # the .close() method when we've finished writing to it.
>>> file_out = open(filnam, 'a')
>>> print('I live in Bikini Bottom.', file=file_out)
>>> file_out.close()
>>>
>>> # Using 'a' is safe because the contents of a file won't be erased
>>> # if it exists.  Instead anything written to the file will be
>>> # appended to the existing file.  If the file doesn't exist, it'll
>>> # be created.
```

Find the file you created on your hard drive and see what's in it. Show it to your TA to get credit for this task.

Task 2: Using `main()` in programs

In some programming languages, e.g., C, the use of a `main()` function is required. While it's not required in Python, most programmers use it. One reason is because it gives another person a basic understanding of how a program works. **`main()` is always a void function with no parameters, and you never use `return` with it.** It's defined after all other functions have been defined. At the very end, you have to call `main()` just as you have to call any other function.

In this task, we'll start with a simple program with just two functions, but first we need a little background information. How many digits are there in 10^1, 10^2, and 10^3? It's fairly easy to answer 2, 3, and 4, respectively, and to convince ourselves that, in general, 10^n is 1 followed by n zeros and, thus, 10^n has $n + 1$ digits. But what about 3^{100}? How many digits are there in 3^{100}?[1] In this task, we'll write a program to answer this question. Counting digits might seem pointless to you, but counting is frequently used in programs (e.g., for bioinformatics applications).

In an IDLE Editor window, write a program that counts the number of digits in j^k where j and k are integer values specified by the user. Your program should consist of the following functions.

- **`calc_digits()`**: Non-void function with two integer parameters, `m` and `n`, that calculates the value of m^n and the number of digits in m^n and returns both calculations.
- **`main()`**: Void function which prompts the user for the base j and exponent k integer values, passes these as arguments to `calc_digits()`, and displays the results of the calculations. **Note that the void function `main()` is the exception to the rule of using `return`. Thus, don't use a `return` statement here or any time you define `main()`.**

Include `docstrings` in each function and end your program with a call to `main()`. Examples are given below for how your program should work.

```
Enter the base j: 3
```

[1]Actually, there's a way to figure this out fairly easily: use `int(`$k\log_{10}(j)$`) + 1`, where $j = 3$ and $k = 100$.

```
Enter the exponent k: 100

3 ** 100 = 515377520732011331036461129765621272702107522001
This number has 48 digits.

Enter the base j: 2
Enter the exponent k: 64

2 ** 64 = 18446744073709551616
This number has 20 digits.
```

Hint: You'll need to use the functions `str()` and `len()`. Recall:

```
>>> a = str(123)          # The argument is an integer
>>> a                     # 'a' is a string!
'123'
>>> len(a)                # 'a' is 3 characters long
3
```

Save your program as `lab6_t2.py` in a folder called Lab6 under CS111, and prove to your TA that it works for credit.

Task 3: Functions with a list argument

In Task 1, we introduced the use of lists with functions. In this task, you'll write two different programs, one similar to the example in Task 1 and the other just a little different. Use the IDLE Editor window for your coding, and save the first program as `lab6_t3a.py` and the second as `lab6_t3b.py`, both in folder Lab6.

Program 1: In this program, you'll write the functions described below. Don't forget to use a header and docstrings, and end your program with a call to `main()`.

- **`add_sort()`**: A void function with one parameter, a list of names. The function prompts a user for a name (see examples below), adds the name to the list, and then sorts the list. It should return nothing (it's a void function!).
- **`main()`**: A void function with no parameters. Create a list of five names of your choosing, call the function `add_sort()` with your list as the argument, and print the sorted list as shown in the examples below. *Don't use* `return`*!*

```
Enter the name you want to add to your list: Svetlana
Your sorted list is ['Abu', 'Armen', 'Jin', 'Saana', 'Svetlana', 'Zhila'].

Enter the name you want to add to your list: Gabe
Your sorted list is ['Abu', 'Armen', 'Gabe', 'Jin', 'Saana', 'Zhila'].
```

Program 2: In this program, you'll write the functions described below. Don't forget to use a header and docstrings, and end your program with a call to `main()`.

- **`add_sum()`**: A non-void function with one parameter, a list of numbers. The function prompts a user for another number (see examples below), adds the number to the list, sums the list, and returns the sum to `main()`.

- **`main()`**: A void function with no parameters. Create a list of five numbers (integers and/or floats of your choosing) using a suitable lvalue. Call the function `add_sum()` with your list as the argument (don't forget an lvalue, but don't use `sum` because it will overwrite the function `sum()`), and print the list and its sum as shown in the examples below.

```
Enter the number you want to add to your list: 123.45
Your list of [2, 3, 5, 7, 11, 123.45] sums to 151.45.

Enter the number you want to add to your list: 13
Your list of [2, 3, 5, 7, 11, 13.0] sums to 41.0.
```

After your programs are running correctly, show them to your TA to get credit.

Task 4: Doomsday algorithm redux

In our last lab, we wrote two functions to find the doomsday. In this task, we'll add two more functions. Recall that a year's doomsday is given by the equation:

$$doomsday = (((yr \mathbin{//} 12) + (yr \mathbin{\%} 12) + (yr \mathbin{\%} 12) \mathbin{//} 4) \mathbin{\%} 7 + anchor) \mathbin{\%} 7$$

where yr is the last two digits of the year and the anchor day is an integer between 0 and 6 with 0 corresponding to Sunday and 6 to Saturday.

For this task, open `lab5_t4.py` in an IDLE Editor window, and save it as `lab6_t4.py` under folder Lab6. Then code the functions described below beneath the two functions you already wrote. Don't forget to edit the header in your program, use docstrings in your functions, and end your program with a call to `main()`.

- **`get_day()`**: A non-void function with one parameter, the doomsday calculated by `calc_doom()`, determines which day of the week it represents (0=Sunday, 6=Saturday), and returns this day to `main()`. Use an `if-elif-else` construct.
- **`main()`**: A void function with no parameters. It prompts the user for a 4-digit year (see examples below), separates the input year into two parts with the first two digits used to determine the anchor and the last two digits used as the year (yr) in the doomsday equation (`divmod(year, 100)` works well for this), calls `det_anchor` which returns the anchor, calls `calc_doom()` which returns the doomsday, and finally calls `get_day()` to determine the day of the week which it then prints as shown below.

Here are some examples for how your program should work.

```
Enter a year [4-digit integer]: 1966
The doomsday for 1966 is Monday.

Enter a year [4-digit integer]: 2023
The doomsday for 2023 is Tuesday.
```

Now you're written a program that has four functions, so you see it isn't really hard to do. Run your program for your TA to get credit.

If you're having trouble understanding functions, please see me or a TA during office hours. Functions are a fundamental part of programming, and we'll be using them for the rest of the semester. Thus, you really need to feel comfortable with them.