CMSC201
Computer Science I for Majors

Lecture 10 – Functions
Last Class We Covered

• Using `for` loops
  – Syntax
  – Using it to iterate over a list
  – Using it for “counting” the number of actions

• The `range()` function
  – Syntax
  – Three forms: one, two, or three numbers
Any Questions from Last Time?
Today’s Objectives

• To learn why you would want to divide your code into smaller, more specific pieces (functions!)
• To be able to define new functions in Python
• To understand the details of function calls and parameter passing in Python
• To use functions to reduce code duplication and increase program modularity
Review: Looping and \texttt{range()}
Review of `range()` Function

```python
for i in range(5):
    print(i)
```

What is the output of this code?

- 0
- 1
- 2
- 3
- 4

Range generates a list of numbers up to (but not including) the number
Review of \texttt{range()} Function

\begin{verbatim}
for i in range(-3, -13, -3):
    print(i)
\end{verbatim}

What is the output of this code?

\begin{itemize}
  \item $-3$
  \item $-6$
  \item $-9$
  \item $-12$
\end{itemize}

With three inputs, we can change the step to a negative to let us count down.
Differences Between the Loops

• Though they are both loops, **for** loops and **while** loops behave very differently

• What does the loop do?
  – **for** loop:
    • Iterate over a list
  – **while** loop:
    • Evaluate a conditional

Even when we use `range()`

What?!

Remember, `range()` creates a list of numbers!
Differences Between the Loops

- What is the syntax of the loop?
  - for loop:
    - for listVariable in listName:
    - Must contain list name and a list variable
  - while loop:
    - while CONDITIONAL (== True):
    - Must use a conditional that contains a variable that changes as the loop is run
Differences Between the Loops

• How is the loop variable updated?
  – **for** loop:
    • The loop itself updates the loop variable
    • First time through, it is element at index 0;
      second time through, element at index 1; etc.
  – **while** loop:
    • Programmer must update the loop variable
    • Updating is not done automatically by Python
Infinite Loops

• An *infinite loop* is a loop that will run forever

• Can we have an infinite loop using *for*?
  – No! The *for* loop goes through a set number of steps (iterating or counting) and will always end

• Can we have an infinite loop using *while*?
  – Yes! The *while* loop’s loop variable is controlled by us, and we can make mistakes
Control Structures (Review)

• A program can proceed:
  – In sequence
  – Selectively (branching): make a choice
  – Repetitively (iteratively): looping
  – By calling a function

[boxed text: focus of today’s lecture]
Introduction to Functions
Functions We’ve Seen

• We’ve actually seen (and used) two different types of functions already!

• Built-in Python functions
  – For example: `print()`, `input()`, casting, etc.

• Our program’s code is contained completely inside the `main()` function
  – A function that we created ourselves
Parts of a Function

use “def” to create a function

```python
def main():
a = 5
print(a)
print(type(a))
main()
```

The output:
```
bash-4.1$ python test.py
5
<class 'int'>
bash-4.1$
```

calls “main”
calls “print” function
calls “type” function
Why Use Functions?

• Functions reduce code duplication and make programs more easy to understand and maintain

• Having identical (or similar) code in more than one place has various downsides:
  1. Don’t want to write the same code twice (or more)
  2. The code must be maintained in multiple places
  3. Code is harder to understand with big blocks of repeated code everywhere
What are Functions?

- **function** is like a subprogram
  - A small program inside of a program
- The basic idea:
  - We write a sequence of statements
  - And give that sequence a name
  - We can then execute this sequence at any time by referring to the sequence’s name
When to Use Functions?

• It’s not an exact science, and there’s no hard rules on when to use functions

• If you have a block of code that performs a specific task, that might make a good function

```python
def debugList( theList ):
    for i in range( len(theList) ):
        print("At index", i, "list is", theList[i])
```
When to Use Functions?

- Functions are used when you have a block of code that you want to be able to:
  - Write only once and be able to use again
    - Example: getting input from the user
  - Call multiple times at different places
    - Example: printing out a menu of choices
  - Differ a little bit when you call it each time
    - Example: printing out a greeting to different people
Function Vocabulary

• Function **definition**:  
  – The part of the program that creates a function  
  – For example: “`def main():`” and the lines of code that are indented inside of `def main():`

• Function **call** (or function invocation):  
  – When the function is used in a program  
  – For example: “`main()`” or “`print("Hello")`”
Function Example
Note: Toy Examples

• The example we’re going to look at today is something called a \textit{toy example}

• It is purposefully simplistic (and kind of pointless) so you can focus on:
  – The concept being taught
  – \textbf{Not} how the code itself works

• Sadly, it has nothing to do with actual toys
“Happy Birthday” Program

• Happy Birthday lyrics...

```python
def main():
    print("Happy birthday to you!")
    print("Happy birthday to you!")
    print("Happy birthday, dear Maya...")
    print("Happy birthday to you!")
```

• Gives us this...

```
bash-4.1$ python birthday.py
Happy birthday to you!
Happy birthday to you!
Happy birthday, dear Maya...
Happy birthday to you!
```
Simplifying with Functions

• Most of this code is repeated (duplicate code)

```python
print("Happy birthday to you!")
```

• We can define a function to print out that line

```python
def happy():
    print("Happy birthday to you!")
```

• Let’s update our program to use this function
Updated “Happy Birthday” Program

• The updated program:

```python
def happy():
    print("Happy birthday to you!")

def main():
    happy()
    happy()
    print("Happy birthday, dear Maya...")
    happy()

main()
```
More Simplifying

• This clutters up our main function, though
• We could write a separate function that sings “Happy Birthday” to Maya, and call it in `main()`

```python
def singMaya():
    happy()
    happy()
    print("Happy birthday, dear Maya...")
    happy()
```
New Updated Program

• The new updated program:

```python
def happy():
    print("Happy birthday to you!")

def singMaya():
    happy()
    happy()
    happy()
    print("Happy birthday, dear Maya...")
    happy()

def main():
    singMaya()  # sing Happy Birthday to Maya
    main()
```
Updated Program Output

```
bash-4.1$ python birthday.py
Happy birthday to you!
Happy birthday to you!
Happy birthday, dear Maya...
Happy birthday to you!
```

Notice that despite all the changes we made to the code, the output is still exactly the same as before
Someone Else’s Birthday

• Creating this function saved us a lot of typing!

• What if it’s Luke’s birthday?
  – We could write a new `singLuke()` function!

```python
def singLuke():
    happy()
happy()
happy()
print("Happy birthday, dear Luke...")
happy()
```
"Happy Birthday" Functions

```python
def happy():
    print("Happy birthday to you!")

def singMaya():
    happy()
    happy()
    print("Happy birthday, dear Maya...")
    happy()

def singLuke():
    happy()
    happy()
    print("Happy birthday, dear Luke...")
    happy()

def main():
    singMaya()  # sing Happy Birthday to Maya
    print()    # empty line between the two
main()
```
Updated Program Output

bash-4.1$ python birthday2.py
Happy birthday to you!
Happy birthday to you!
Happy birthday, dear Maya...
Happy birthday to you!
Happy birthday to you!
Happy birthday to you!
Happy birthday, dear Luke...
Happy birthday to you!
bash-4.1$
Multiple Birthdays

• This is much easier to read and use!
• But... there’s still a lot of code duplication

• The only difference between singMaya() and singLuke() is what?
  — The name in the third print() statement
• We could combine these two functions into one by using something called a parameter
Function Parameters
What is a Parameter?

• A **parameter** is a variable that is initialized when we call a function

• We can create a generic `sing()` function that takes in a person’s name as a parameter

```python
def sing(person):
    happy()
    happy()
    print("Happy birthday, dear", person + "...")
happy()
```
"Happy Birthday" with Parameters

```python
def happy():
    print("Happy birthday to you!")

def sing(person):
    happy()
    happy()
    happy()
    print("Happy birthday, dear", person + "...")
    happy()

def main():
    sing("Maya")
    print()
    sing("Luke")
main()
```
“Happy Birthday” with Parameters

```python
def happy():
    print("Happy birthday to you!")

def sing(person):
    happy()
    happy()
    print("Happy birthday, dear", person + "...")
    happy()

def main():
    sing("Maya")
    print()
    sing("Luke")
main()
```
Updated Program Output

bash-4.1$ python birthday3.py
Happy birthday to you!
Happy birthday to you!
Happy birthday, dear Maya...
Happy birthday to you!

Happy birthday to you!
Happy birthday to you!
Happy birthday, dear Luke...
Happy birthday to you!
bash-4.1$

This looks the same as before!
That’s fine! We wanted to make our code easier to read and use, not change the way it works.
Exercise: Prompt for Name

• How would we update the code in `main()` to ask the user for the name of the person?
  – Current code looks like this:

```python
def main():
    sing("Maya")
main()
```
Solution: Prompt for Name

- How would we update the code in `main()` to ask the user for the name of the person?
  - Updated code looks like this:

```python
def main():
    birthdayName = input("Whose birthday? ")
    sing(birthdayName)
main()
```

Nothing else needs to change – and the `sing()` function stays the same
Exercise Output

bash-4.1$ python birthday4.py
Whose birthday? UMBC
Happy birthday to you!
Happy birthday to you!
Happy birthday, dear UMBC...
Happy birthday to you!
bash-4.1$
How Parameters Work
Functions and Parameters

• Each function is its own little subprogram
  – Variables used inside of a function are *local* to that function
  – Even if they have the same name as variables that appear outside that function

• The *only* way for a function to see a variable from outside itself is for that variable to be passed as a *parameter*
Function Syntax with Parameters

• A function definition looks like this:

```python
def fxnName(formalParameters):
    # body of the function
```

- function name: follows same syntax rules as variable names
  (no special characters, can’t start with a number, no keywords, etc.)

- the formal parameters that the function takes in – can be empty!
Formal Parameters

• The *formal parameters*, like all variables used in the function, are *only* accessible in the body of the function

• Variables with identical names elsewhere in the program are distinct from those inside the function body
  
  – We often call this the “scope” of a variable
Scope

• If variables are boxes, then a function and its variables are like a pallet that holds boxes.
Scope

• If variables are boxes, then a function and its variables are like a pallet that holds boxes
  
Can hold unique values
Example of Scope

• This is our president, Freeman A. Hrabowski III
  – According to Wikipedia, he is a “a prominent American educator, advocate, and mathematician” and has been the President of UMBC since 1992
  – He will also take you up to the roof of the Admin building to show off the campus (it’s super cool)
Example of Scope

• This is my (fictional) dog, a Chesapeake Bay Retriever also named Hrabowski
  – He is super cute, knows tons of tricks, and his favorite toy is his squeaky yellow duck
  – He also loves to spin in circles while chasing his tail
Example of Scope

• We have two very different things, both of which are called Hrabowski:
  – UMBC’s President Hrabowski
  – My (fictional) dog Hrabowski

• If you go outside this classroom and tell someone “Hrabowski loves to chase his tail, it’s super cute” they will be very confused
Example of Scope

• In the same way, a variable called `person` inside a function like `sing()` is a completely different variable from `person` in `main()`

• The `sing()` function has one idea of what the `person` variable is, and `main()` has another

• It depends on the context, or “scope” we are in
Calling Functions with Parameters
Calling with Parameters

• In order to call a function with parameters, use its name followed by a list of variables

  myFunction("my string", 17)

• These variables are the *actual parameters*, or *arguments*, that are passed to the function
Python and Function Calls

• When Python comes to a function call, it initiates a four-step process:
  1. The calling program *suspends execution* at the point of the *call*
  2. The *formal parameters* of the function get assigned the values supplied by the *actual parameters* in the call
  3. The body of the function is *executed*
  4. *Control* is returned to the point *just after* where the function was called
Code Trace: Parameters

• Let’s trace through the following code:

```python
sing("Maya")
print()
sing("Luke")
```

• When Python gets to the line `sing("Maya")`, execution of `main` is temporarily suspended

• Python looks up the definition of `sing()` and sees it has one formal parameter, `person`
Code Trace: Parameters

```python
def happy():
    print("Happy birthday to you!")

def sing(person):
    happy()
    happy()
    print("Happy birthday, dear", person + "...")
    happy()

def main():
    sing("Maya")
    print()
    sing("Luke")

main()
```
Initializing Formal Parameters

• The *formal parameter* is assigned the value of the *actual parameter*

• When we call `sing("Maya")`, it as if the following statement was executed in `sing()`

  ```
  person = "Maya"
  ```
Code Trace: Parameters

• Next, Python begins executing the body of the `sing()` function
  – First statement is another function call, to `happy()` – what does Python do now?
    – Python suspends the execution of `sing()` and transfers control to `happy()`
    – The `happy()` function’s body is a single `print()` statement, which is executed
  – Control returns to where it left off in `sing()`
Code Trace: Parameters

• Execution continues in this way with two more “trips” to the `happy()` function.

• When Python gets to the end of `sing()`, control returns to...
  – `main()`, which picks up...
  – where it left off, on the line immediately following the function call.
def main():
    sing("Maya")
    print()
    sing("Luke")

def sing(person):
    happy()
    happy()
    print("Happy BDay ", person)
    happy()

Note that the person variable in sing() disappeared after we exited the function!
Local Variables

• When a function exits, the local variables (like `person`) are deleted from memory

• If we call `sing()` again, a new `person` variable will have to be re-initialized
  – Local variables do **not** retain their value between function executions
Code Trace: Parameters

• Next statement in `main()` is the empty call to `print()`, which simply produces a blank line

• Python sees another call to `sing()`, so...
  – It suspends execution of `main()`, and...
  – Control transfers to...
    - the `sing()` function
  – With the actual parameter...
    "Luke"
def main():
    sing("Maya")
    print()
    sing("Luke")

def sing(person):
    happy()
    happy()
    print("Happy BDay", person)
    happy()

    person: "Luke"

• The body of `sing()` is executed with the argument "Luke"
  – Including its three side trips to `happy()`
• Control then returns to `main()`
Multiple Parameters
Multiple Parameters

• One thing we haven’t discussed is functions with *multiple parameters*

• When a function has more than one parameter, the formal and actual parameters are matched up based on **position**
  
  – First actual parameter becomes the first formal parameter, etc.
Multiple Parameters in `sing()`

- Let’s add a second parameter to `sing()` that will take in the person’s age as well
- And print out their age in the song

```python
def sing(person, age):
    happy()
    happy()
    print("Happy birthday, dear", person, "...")
    print("You're", age, "years old now...")
    happy()
```
Multiple Parameters in `sing()`

• What will happen if we use the following call to the `sing()` function in `main()`?

```python
def main():
    sing("Maya", 46)

main()
```

• It will print out:

Happy birthday to you!
Happy birthday to you!
Happy birthday, dear Maya...
You're 46 years old now...
Happy birthday to you!
Assigning Parameters

• Python is simply assigning the first actual argument to the first formal argument, etc.

```python
sing("Maya", 46) # function call

def sing(person, age):
    # function body goes here
```
Parameters Out-of-Order

• What will happen if we use the following call to the `sing()` function in `main()`?

```python
def main():
    sing(46, "Maya")
main()
```

• It will print out:

```
Happy birthday to you!
Happy birthday to you!
Happy birthday, dear 46...
You're Maya years old now...
Happy birthday to you!
```
Parameters Out-of-Order

• Python isn’t smart enough to figure out what you meant for your code to do
  – It only understands the exact code

• That’s why it matches up actual and formal parameters based only on their order
Practice Problems

• Write a function called `avg()` that takes in a list of numbers, calculates the average, and prints the result to the screen.

• Write a function that prints out the lyrics to a song – you can pick any song you like!
  – If it has a chorus, put that in a separate function, and call it when necessary.
Announcements

• Homework 4 is due Wednesday
  – Homework 2 grades went out Tuesday night

• Homework 5 comes out this week

• The midterm exam is fast approaching...
  – During your regular class on October 19th and 20th!
  – We’ll review in class on the 17th and 18th