CMSC201
Computer Science I for Majors

Lecture 10 – Functions

Prof. Katherine Gibson
Prof. Jeremy Dixon

Last Class We Covered

- Using **`while`** loops
  - Syntax of a **`while`** loop
  - Interactive loops
- Mutating a list
  - **`append()`**
  - **`remove()`**
- Nested loops

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

Control Structures (Review)

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

focus of today's lecture

Introduction to Functions

Functions We've Seen

- We've actually seen (and been using) two different types of functions already!
  - Our program's code is contained completely inside the **`main()`** function
  - Built-in Python functions
    - For example: **`print()`**, casting, etc.

Parts of a Function

use "**def**" to create a function

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

function body

calls "**print**" function

calls "**type**" function

calls "**main**"

The output:

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

Why Use Functions?

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

What are Functions?

- A 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 execute this sequence at any time by referring to the sequence's name

Function Vocabulary

- Function **definition**:
  - The part of the program that creates a function
  - For example: "`def main():`"
- Function **call** (or function invocation):
  - When the function is used in a program
  - For example: "`main()`" or "`print("Hello")`"

Example Function

“Happy Birthday” Program

- Happy Birthday lyrics…

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

Simplifying with Functions

- A lot of this code is repeated (duplicate code)

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

- We can ***define*** a function to print out that line

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

- We can update our program to use this function

Updated “Happy Birthday” Program

- The updated program:

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

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

More Simplifying

- Even this version is a bit repetitive
- We could write a separate function that sings "Happy Birthday" to Fred, and call it in **main()**

```
def singFred():
    happy()
    happy()
    print("Happy birthday, dear Fred...")
    happy()
```

New Updated Program

- The new updated program:

```
def happy():
    print("Happy birthday to you!")
def singFred():
    happy()
    happy()
    print("Happy birthday, dear Fred...")
    happy()
def main():
    singFred() # sing Happy Birthday to Fred
main()
```

Updated Program Output

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

Someone Else's Birthday

- Creating this function saved us a lot of typing!
- What if it's Lucy's birthday?
  - We could write a new **singLucy()** function!

```
def singLucy():
    happy()
    happy()
    print("Happy birthday, dear Lucy...")
    happy()
```

"Happy Birthday" Functions

```
def happy():
    print("Happy birthday to you!")
def singFred():
    happy()
    happy()
    print("Happy birthday, dear Fred...")
    happy()
def singLucy():
    happy()
    happy()
    print("Happy birthday, dear Lucy...")
    happy()
def main():
    singFred() # sing Happy Birthday to Fred
    print()    # empty line between the two
    singLucy() # sing Happy Birthday to Lucy
main()
```

Updated Program Output

```
bash-4.1$ python birthday2.py
Happy birthday to you!
Happy birthday to you!
Happy birthday, dear Fred...
Happy birthday to you!

Happy birthday to you!
Happy birthday to you!
Happy birthday, dear Lucy...
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 **`singFred()`** and **`singLucy()`** is …
  - the name in the third **`print()`** statement
- We could combine these two functions 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

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

parameter

"Happy Birthday" with Parameters

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

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

def main():
    sing("Fred")
    print()
    sing("Lucy")
main()
```

“Happy Birthday” with Parameters

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

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

def main():
    sing("Fred")
    print()
    sing("Lucy")
main()
```

parameter passed in

parameter being used

function call with parameter

function call with parameter

Updated Program Output

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

Happy birthday to you!
Happy birthday to you!
Happy birthday, dear Lucy...
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:

```
def main():

    sing("Fred")
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:

```
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:

function name: follows same syntax rules as variable names

(no special characters, can't start with a number, no keywords, etc.)

```
def fnxName(formalParameters):
    # body of the function
```

the formal parameters that the function takes in – **can be empty!**

Formal Parameters

- The *formal parameters*, like all variables used in the function, are **<u>only</u>** accessible in the body of the function

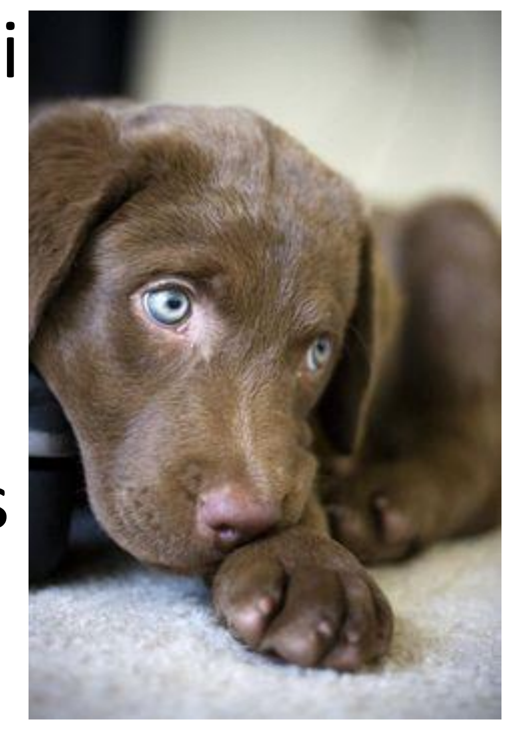
- Variables with identical names elsewhere in the program are distinct from those inside the function body
  - We often call this the "*scope*" of a variable

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 likes to beg for scraps from the dinner table
  - 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 <u>very</u> confused

Example of Scope

- In the same way, a variable called **`person`** inside a function like **`sing()`** is a completely <u>different</u> 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:

```
sing("Fred")
print()
sing("Lucy")
```

- When Python gets to the line **`sing("Fred")`**, execution of **`main`** is temporarily suspended
- Python looks up the definition of **`sing()`** and sees it has one formal parameter, **`person`**

Code Trace: Parameters

```
def happy():
    print("Happy birthday to you!")
def sing(person):
    happy()
    happy()
    print("Happy birthday, dear", person + "...")
    happy()

def main():
    sing("Fred")
    print()
    sing("Lucy")

main()
```

formal parameter

actual parameter

actual parameter

Initializing Formal Parameters

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

- When we call **`sing("Fred")`**, it as if the following statement was executed in **`sing()`**

```
person = "Fred"
```

Visual Code Trace

```
def main():
    sing("Fred")
    print()
    sing("Lucy")
```

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

Visual Code Trace

```
def main():
    sing("Fred")
    print()
    sing("Lucy")
```

person = "Fred"

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

person: "Fred"

Note that the variable **`person`** has been initialized in **`sing()`**

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()**

Visual Code Trace

```
def main():
    sing("Fred")
    print()
    sing("Lucy")
```

person = "Fred"

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

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

person: "Fred"

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

Visual Code Trace

```
def main():
    sing("Fred")
    print()
    sing("Lucy")
```

person = "Fred"

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

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

Note that the **person** variable in **sing()** disappeared!

Local Variables

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

- If we call **`sing()`** again, **`person`** 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...

    "Lucy"

Visual Code Trace

```
def main():
    sing("Fred")
    print()
    sing("Lucy")
```

person = "Lucy"

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

person: "Lucy"

The body of **sing()** is executed with the argument "Lucy"

- 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

```
def sing(person, age):
    happy()
    happy()
    print("Happy birthday, dear", person, "...")
    print("You're already", age, "years old...")
    happy()
```

Multiple Parameters in **sing()**

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

```
def main():
    sing("Fred", 46)
main()
```

- It will print out:

```
Happy birthday to you!
Happy birthday to you!
Happy birthday, dear Fred...
You're only 46 years old...
Happy birthday to you!
```

Assigning Parameters

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

```
sing("Fred", 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()**?

```
def main():
    sing(46, "Fred")
main()
```

- It will print out:

```
Happy birthday to you!
Happy birthday to you!
Happy birthday, dear 46...
You're only Fred years old...
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

Any Other Questions?

Announcements

- Homework 6 is out
  - Problems to help review for the midterm
  - Due by Tuesday (March 8th) at 8:59:59 PM
    - Questions are difficult – start early!
- Midterm is March 9th/10th
  - Review in class on March 7th/8th
  - Bring paper and pencil to in-class review

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 called **`pali()`** that takes in a string and prints out whether it is a palindrome; you can use your code from lab as the basis for the function body
- Write a function that prints out the lyrics to a song – you can pick any song you like!