Computer Science

Summer Springboard

SUMMER SPRINGBOARD
Look Inward. Go Upward

Day 4: (Fun)ctions

Project One Check-In

Remember that project one presentations are tomorrow! Before we get started today, take some time to discuss the following questions in groups or as a class:

- What is your idea for your first project?
- Have you started? If so, what have you done so far?
- Is there anything you are struggling with?

Functions

Functions

- The main focus of today will be on functions!
- Functions are essential to many applications of coding and are an integral part of most popular programming languages.
- Functions are like individual recipes in a cookbook. Each function is a self-contained set of instructions designed to perform a specific task.

SUMMER SPRINGBOARD
Look Inward. Go Upward

Relating it Back to Math

- In math, a function is a relation between a set of inputs and a set of outputs, where each input is related to exactly one output.
- In Python this is relaxed a bit because an input is allowed to have no inputs, no outputs, multiple outputs, but it is largely the same concept
    - Your function has a name, it can take some set of inputs, and gives you some output based on how it processes the input (if any).

Return Values: The Outcome

- Functions can send Python objects back to the caller using the **return** statement.
- This feature turns functions into versatile tools, not just limited to performing actions, but also returning results

The Blueprint of a Function

- To define a function, use the **def** keyword followed by a descriptive name, your input variables in parentheses, and then a colon. Any code that is in your function should be indented.
- Here is the basic template:

```
def function_name(input_1, input_2, ...):
    doSomething
    return result  # optional
```

Example: Math -> Python

- A function example in math: $f(x) = x^2 + x - 2$
- Translated into Python:

```
def f(x):
    result = (x ** 2) + x - 2
    return result
```

- In both cases, the function is named **f** and takes in an input, **x**, and gives an output based on the input.

Calling a Function

- Bring your function to life by calling its name with the necessary parameters.

For example, if we have a function, **greet**, as defined below:

```
def greet(name):
    print(f"Hello, {name}!")  # formatted string*
```

We can call **greet** like so: **greet("Alice")**, which results in "Hello, Alice" being printed.

* Formatted strings allow for the inclusion of expressions inside string literals using **{}**, making string contacts concise.

Parameters vs Arguments

- Parameters are the variables that are listed inside the parentheses in the function definition. They act as placeholders for the values the function needs.
- Arguments: values passed into the function when it is called

So in this example:

```
def f(x):
    result = (x ** 2) + x - 2
    return result
ans = f(2)
```

The **x** in **f(x)** is the parameter and the **2** in **f(2)** is the argument

Back to Reality

- Think of functions as your go-to kitchen gadgets. Just like how a blender or a toaster is used for specific tasks, a function in Python is crafted for particular operations, making the overall process (or program) more efficient and orderly.

Scope

SUMMER
SPRINGBOARD
Look Inward. Go Upward.

Scope of Variables: Knowing the Boundaries

- In Python, scope determines the visibility of variables.
- If you were on a boat in the ocean, some islands would be more visible from your current position and the most visible are within your current scope.

Types of Scope: The Islands of Code

- Local Scope: The nearby island. Variables created inside a function belong to the local scope of that function and can only be accessed within the function.
- Global Scope: The entire archipelago. Variables created in the main body of the Python script are global and can be accessed anywhere in the script.

The Local Scope: Private Islands

- Created when a function is called, destroyed when the function finishes.
- Example: A variable defined within a function to store intermediate results

The Global Scope: Public Beaches

- Created when the script starts, destroyed when the script ends.
- Example: A variable defining a constant used throughout the script.

Navigating the Waters: ‘global’ Keyword

- Use the **global** keyword to modify a global variable inside a function.
- Example:

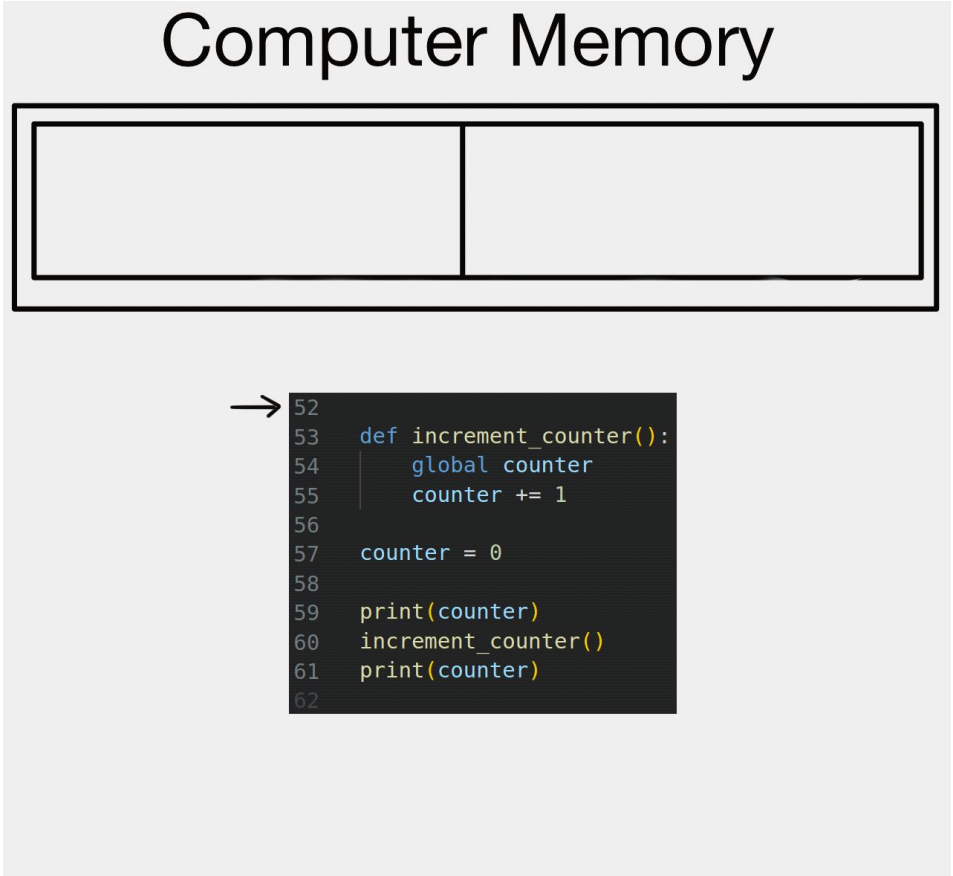
```
54  counter = 0
55  def increment_counter():
56      global counter
57      counter += 1
```

SUMMER SPRINGBOARD
Look Inward. Go Upward

Best Practices: Responsible Sailing

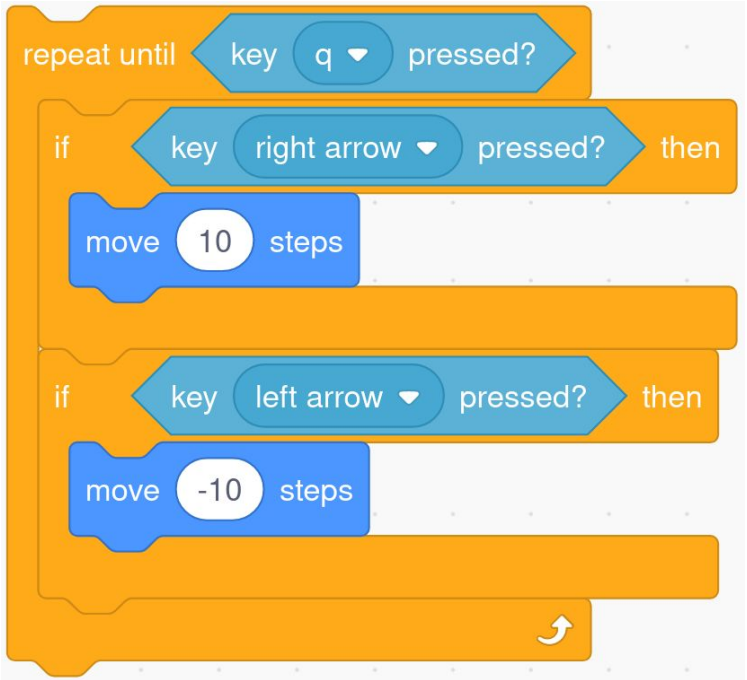
- Prefer local scope to avoid unintended side-effects on global variables.
- Use global variables sparingly, mostly for constants or shared data.
- Python follows the LEGB rule for scope resolution: Local, Enclosing, Global, Built-in.
    - This means that if there is are variables with the same name in different scopes, Python will resolve the names in this order.

Animation: Functions



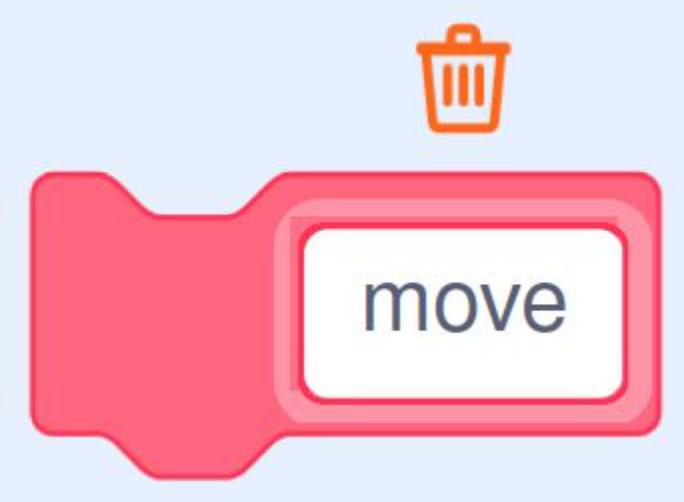
Functions in Scratch

- First, let's edit our Scratch code to make it more useful in the future:



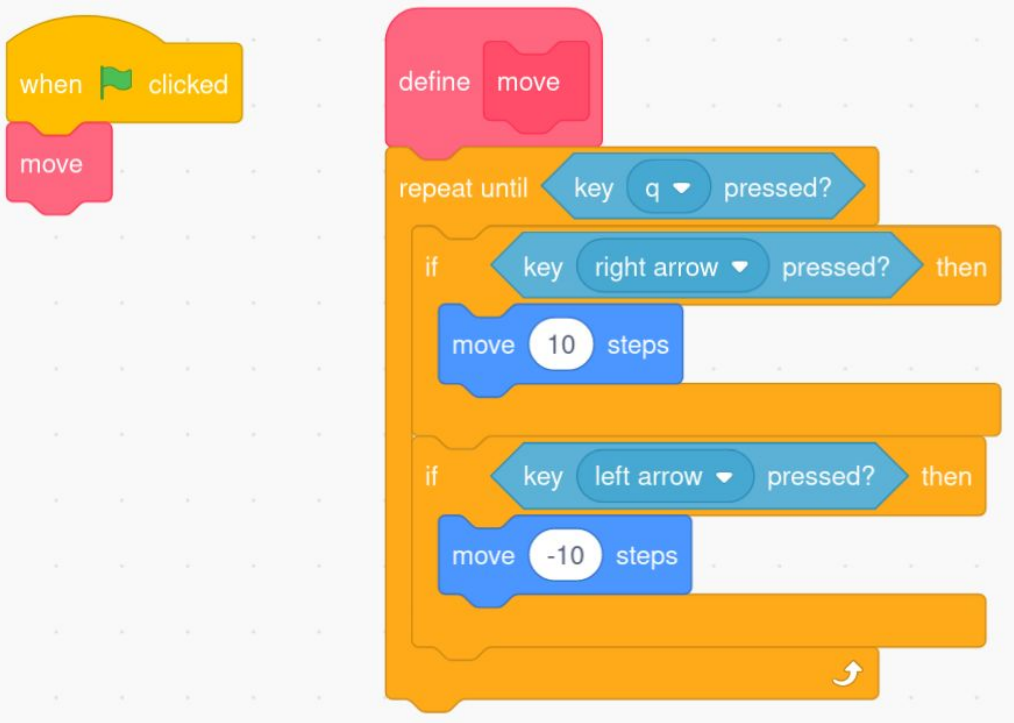
Functions in Scratch (Cont'd)

- Now, go to "My Blocks" and press "Make a Block"
  - A "block" is essentially a function.
  - Name your new block "move"



Functions in Scratch (Cont'd)

- Rearrange your code so that it looks like the below:



Functions in Scratch (Cont'd)

- Now, when the green flag is pressed, we move into our **move** code block, which lets us move around with the left / right arrow keys until **q** is pressed
- Take 10-20 minutes to try to make your own block! See what creative ideas you can come up with.

Fun with Functions

- Now, let's do some practice with functions! Hop over to today's Google Colab notebook and work until the first "***PAUSE***"

SUMMER SPRINGBOARD
Look Inward. Go Upward

Debugging

SUMMER
SPRINGBOARD
Look Inward. Go Upward.

What is ‘debugging’?

- Nobody gets everything right the first time around. Just like anything else in life, sometimes when you start to code, you don’t realize you are doing something wrong until later down the line. Thankfully, in coding you can go back and fix your mistakes so that everything works correctly!
- Crafting Clarity: Debugging is the process of identifying and resolving errors or 'bugs' in your code. It's like being a detective in your own program, searching for clues to fix issues.

Quick Pit-Stop: What's with the term 'bug'?

- The term "bug" to describe technical glitches dates back to the early 19th century, long before the advent of modern computers. Originally, it referred to defects in mechanical systems.
- The term gained legendary status in computer programming due to an incident involving Rear Admiral Grace Hopper, a pioneer in computer science.
- In 1947, while working on the Harvard Mark II computer, her team found a moth causing a malfunction. They literally "debugged" the system by removing the moth.
- Hopper noted this in the log book as a "bug" being found, giving a physical reality to the term.

Common Types of Bugs

- Syntax Errors: Mistakes in the code's syntax.
  - Missing parentheses, typos, etc
- Runtime Errors: Errors that occur while the program is running.
  - Ex: attempting to divide by zero
- Logical Errors: These are the hardest to spot. The code runs but does not produce the expected outcome, often due to errors in logic or algorithm.

Strategies for Effective Debugging

- Read the error messages
- Use print statements
- Break down the code
- Use a debugger

Reading the Error Messages

- Error messages are like the first clues in a detective's casebook. They often point directly to the source of the problem.
- Information in error messages includes: type of error, error description, and location of the error (line number or place in the code where the error was detected)
- Example: **TypeError: unsupported operand type(s) for +: 'int' and 'str'**
    - This error message tells you that you're trying to add an integer to a string, which is not allowed.
- Use the information in the message to trace back to the problematic part of your code and relate the error to the code's context for a better understanding of the cause.

Debugging with Print Statements

- Print statements can illuminate how your code is behaving at specific points.
- Use print statements to track variable values at different stages of your code, helping to track down where things go awry.
- Insert print statements to understand the flow of execution, especially within loops and conditionals.
- Place print statements before and after suspected problem areas to compare expected versus actual outcomes.
- Format print statements clearly and remove them after debugging.

Breaking Down the Code

- Isolate sections of your code to identify where the problem lies
- Run small, manageable code segments independently to check their functionality.
- Gradually expand the tested code until you find the source of the bug.
- Reduce complexity to understand the basic issue.

Using a Debugger

- A debugger is a tool that allows developers to inspect and control the execution of their program line by line, making it invaluable for identifying, diagnosing, and fixing bugs or errors in the code.
- Python has a debugger, **pdb**, built-in.
  - While in the debugging session, you can print variable values, evaluate expressions, and even change variables.
  - Offers precise control over program execution.
- Learning the basic **pdb** commands will give you a much smoother debugging experience.

Patience and Persistence

- Debugging is a skill that comes with much time and practice.
- Initially, using a debugger might seem slow and cumbersome, but with practice, it becomes an invaluable part of the development process.
- And that’s why…

We Have More Practice Problems!

- Let’s go back to the Google Colab notebook and finish the rest of our practice problems for the day!

SUMMER SPRINGBOARD
Look Inward. Go Upward