

Functions

Overview

In this lesson, students will learn how to iterate over lists or until a condition has been met.

Duration

120 minutes

Learning Objectives

In this lesson, students will:

- Define functions to encapsulate blocks of code.
- Use parameters in a function.
- Understand how to return a value from a function.
- Create functions that include loops and conditional logic to generate specific return values.



Pre-Class Materials and Preparation

For remote classrooms: Virtual breakout rooms and Slack may be needed to facilitate the partner exercise and discussions. As you plan for your lesson:

- Consider how you'll create pairs for the partner exercise (randomly, or with pre-assigned partners).
- Determine how (if at all) exercise timing may need to be adjusted.
- For helpful tips, keep an eye out for the **For remote classrooms** tag in the speaker notes.
- Prepare screenshots and answers to exercises in advance so that they can be easily shared in Slack during your lecture.

Pre-Class Materials and Preparation (Cont.)

Pre-Work Review: Many of the concepts covered in this class will recap what students learned in the pre-work; in particular, these myGA lessons:

- Python Functions

We recommend reviewing the pre-work lessons to understand what students will come to class knowing. [Click here](#) to see the study guides for these lessons. You can also view the pre-work materials on myGA.

For review or extension topics, review the notes in the pre-work for recommendations on how to teach the concept. Leverage the discussions, exercises, and knowledge checks to gauge students' understanding of concepts. You can adjust your approach to the lesson overall based on how students do with these exercises.



Suggested Agenda

Time	Activity
0:00–0:15	Welcome + Introductions
0:15–0:50	Python Functions
0:50–1:00	Break
1:00–1:40	Problem-Solving Functions
1:40–1:50	Functions and Lists
1:50–2:00	Wrapping Up, Q&A, and Exit Ticket Completion



Jupyter Notebook

The exercises referenced in this lesson can be found in the [Python Workbooks + Data](#) folder.

Our first few lessons begin by reviewing the notebook, as the same material was covered in the pre-work.

If students feel that they are able to confidently solve the challenges in the workbook, you can quickly skim through the lesson content and have them only complete the challenges.

— Functions

Today's Learning Objectives

In this lesson, you will:

- Define functions to encapsulate blocks of code.
- Use parameters in a function.
- Understand how to return a value from a function.
- Create functions that include loops and conditional logic to generate specific return values.



What We'll Practice Today

This class is a **blended learning experience**. It connects to and reinforces topics that you encountered in the myGA pre-work.

We're going to return to topics covered in the pre-work and build upon them:

- **Defining and invoking functions**
- **Writing pseudocode**





Solo Exercise:

Jupyter Notebook Review

10 minutes



Let's dive right in and use what we learned in the pre-work! We want to understand where you are in your learning journey so that we can give the best possible experience in class.

Look over the exercises in today's Jupyter Notebook and attempt any that seem immediately doable to you.

Then, **rate your confidence level** on today's subjects from 1–5.

Functions



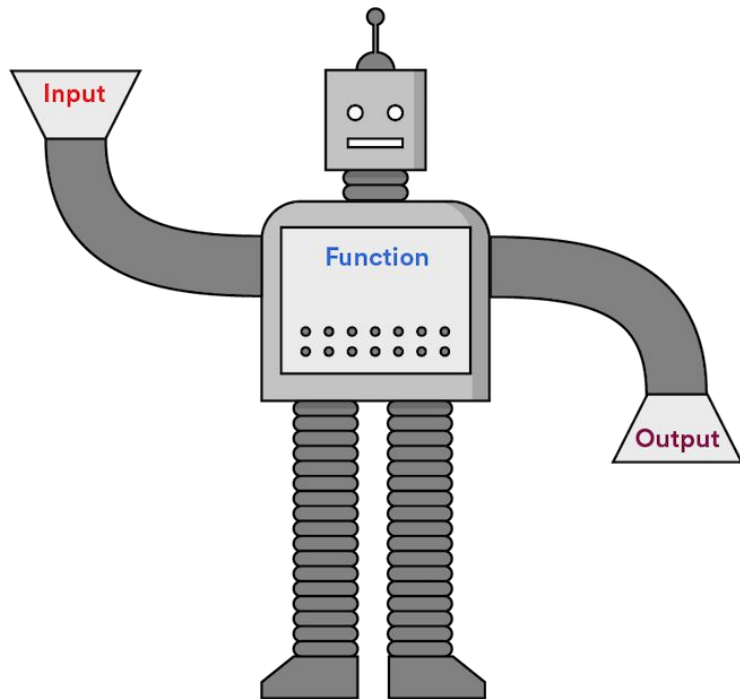
Python Functions



Functions

Functions are chunks of code that are grouped and will execute together, like a modular program within a program.

A function takes input, performs logic, and returns output.



Defining Functions

`def` required

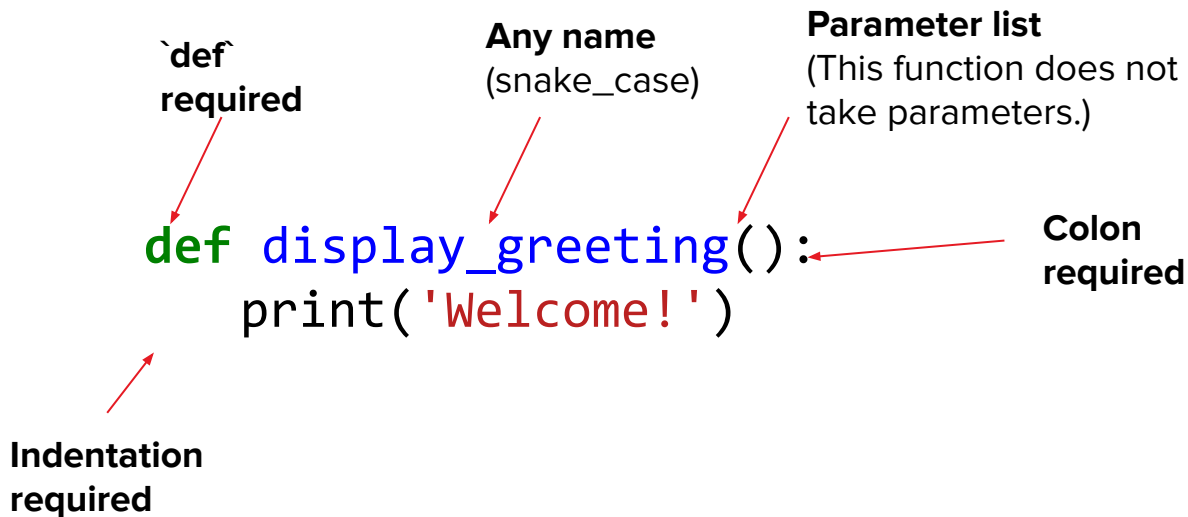
Any name
(snake_case)

Parameter list
(This function does not take parameters.)

Colon required

```
def display_greeting():  
    print('Welcome!')
```

Indentation required

A diagram illustrating the syntax of a Python function definition. The code snippet is: `def display_greeting():
 print('Welcome!')`. Red arrows point from text annotations to specific parts of the code:

- An arrow from **`def` required** points to the `def` keyword.
- An arrow from **Any name (snake_case)** points to the function name `display_greeting`.
- An arrow from **Parameter list (This function does not take parameters.)** points to the empty parentheses `()`.
- An arrow from **Colon required** points to the colon `:` at the end of the first line.
- An arrow from **Indentation required** points to the indentation of the `print` statement on the second line.

Invoking Functions

Defining a function simply sets things up. For anything to happen, you must **invoke** the function at some point.

```
def greetings():
```

```
    print("hello!")
```

```
# Invoke the function using its name and more parentheses.
```


```
greetings()
```



Function Parameters and Arguments

```
get_total_price(100., 0.1)
```

```
def get_total_price(list_price, tax_rate):  
    total_price = list_price * (1. + tax_rate)  
    return total_price
```

Two red arrows originate from the arguments '100.' and '0.1' in the function call above. The first arrow points to the parameter 'list_price' in the function definition. The second arrow points to the parameter 'tax_rate' in the function definition.

Arguments

Values passed in to the function

Parameters

Variable names within the function

Inside the function body:

```
list_price = 100.
```

```
tax_rate = 0.1
```

parameter = argument

Keyword and Positional Arguments

When providing arguments to a function, you can either rely on the position or on keywords to define the specific inputs.

```
def calculate_area(length, width):
```

```
    return length * width
```

```
calculate_area(8,9)
```

```
calculate_area(width=9, length=8)
```





Discussion:

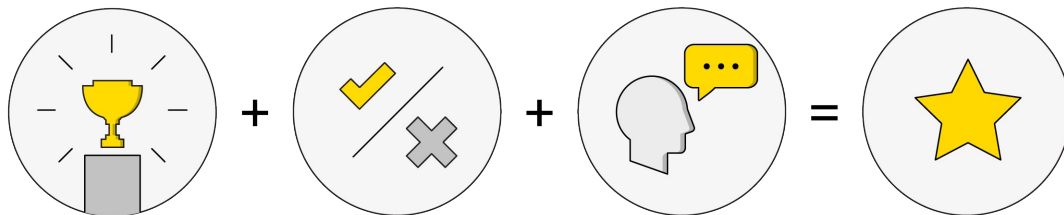
What Does This Function Do?

```
def mystery_function(list_to_check, value):  
    for i in range( len(list_to_check) ):  
        if(list_to_check[i] == value):  
            return i  
  
    return -1
```




Most code challenges you come across will involve writing a function that provides a solution to a given problem.

Section 5.1 has some good initial code challenges to help you practice writing functions.



Functions



Pseudocoding





The **good** news about **computers** is that **they** **do what you tell them to do**. The **bad** news is that **they do what you tell them to do**.

— Ted Nelson, technology pioneer



Thinking Like a Computer

The art of programming requires understanding how a computer thinks:

It only knows what you tell it...

...but it will remember what it's been told.

It only understands a very limited set of phrases (syntax)...

...but you can teach it a lot by combining these basic phrases together.

It will always do what you say...

...but not necessarily what you meant.

It has no understanding of context...

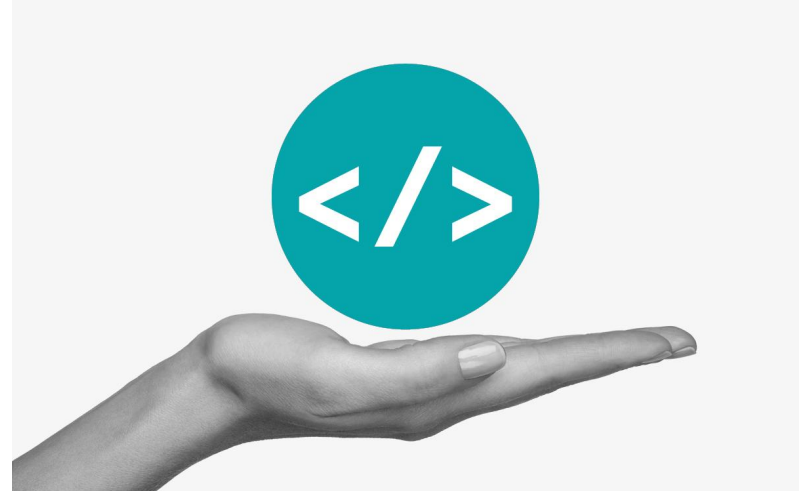
...but it's not shy about saying when it doesn't understand you (error messages).



Using Pseudocode

Pseudocode uses the structure of code without having to worry about the actual coding language (syntax) of the instructions.

Pseudocode is an attempt to break down a larger process into the smallest imaginable component steps.



A Stir Fry Program

```
spices = soy_sauce + rice_vinegar +  
sugar  
oil = 1.5 ounces  
brown_rice = 1.5 cups  
broccoli = 2 cups  
shrimp = .5 pounds
```

```
stir_fry():  
    cook brown_rice  
    whisk spices  
    heat oil in pan  
    add broccoli  
    cook shrimp  
    add spices
```

A Stir Fry Program With Parameters

Recipe verbs are **functions** you can use with the ingredient variables.

The ingredients act as **variables** being passed into the **stir_fry()** **function**.

```
spices = soy_sauce +  
rice_vinegar + sugar  
oil = 1.5 ounces  
brown_rice = 1.5 cups  
broccoli = 2 cups  
shrimp = .5 pounds
```

```
stir_fry(item1, item2, item3, item4, item5):  
    cook(item1)  
    whisk(item2, item3)  
    heat(item3)  
    combine(item3, item4)  
    cook(item5)  
    add(item2)
```

```
stir_fry(brown_rice, spices, oil, broccoli,  
shrimp)
```

**Programs don't get more complicated
because of **code**. They get more complicated
because of the **logic** behind them.**





Discussion:

Pseudocoding a Process

5 minutes



Think about a process that you're required to perform in your role. How you would pseudocode instructions for a new employee?

1. What inputs are required to perform the process?
2. What are the concrete steps involved?
3. What is the output of that process, if any?



The next few functions will require a bit of problem-solving logic to produce a specific result from given inputs.

Pseudocoding may help you by writing out the steps before coding them!



Functions



Functions + Lists



Using map()

It's common to want to apply a function to everything in a list. It's actually so common that there's a method to do exactly that: `map()`!

```
greetings = ["hi", "hello", "salutations"]
```

```
def excitement(word)
```

```
    return upper(word) + "!!!"
```

```
excited_greetings = map(excitement, greetings)
```

```
# excited_greetings is ["HI!!!", "HELLO!!!", "SALUTATIONS!!!"]
```



Lambda Functions

In this and in similar situations, you sometimes don't even want to create a separate function to pass in. You can define a **lambda** function directly in-line by using the lambda keyword and shortened definition syntax:

```
greetings = ["hi", "hello", "salutations"]
```

```
excited_greetings = map(lambda word : upper(word) + "!!!", greetings)
```

```
# excited_greetings is ["HI!!!", "HELLO!!!", "SALUTATIONS!!!"]
```



Functions



Wrapping Up



Recap

In today's class, we...

- Defined functions to encapsulate blocks of code.
- Used parameters in a function.
- Understood how to return a value from a function.
- Created functions that include loops and conditional logic to generate specific return values.

Looking Ahead

On your own:

- Ensure that you've completed the Python pre-work and pre-work quiz.
- Start thinking about capstone project ideas!

Next Class:

Scripting and Modules



Don't Forget: Exit Tickets!



