

TEALS Program

[Home](#) | [Curriculum Map](#) | [Additional Readings](#) | [Change Log](#)

1 A 2nd semester follow-up to the TEALS Intro CS course

- GitHub: <https://github.com/TEALSK12/2nd-semester-introduction-to-computer-science>

The following has been deprecated after the 2019/20 school year:

- GitBook: <https://tealsk12.gitbooks.io/2nd-semester-introduction-to-computer-science/content/>

1.1 About this curriculum

Welcome to the TEALS Intro to Computer Science Second Semester Curriculum. This curriculum is intended for use by TEALS classrooms teaching Introduction to Computer Science in a yearlong format. We expect that students have completed the content from the [1st semester course](#) prior to this curriculum.

1.2 Associated Readings

We have included with this curriculum Associated Readings to dive deeper into topics specific to the instruction of this course. These readings have been adapted from "Think Python: How to Think Like a Computer Scientist" by Allen B. Downey. ([HTML Version](#))([PDF Version](#)). They are specifically referenced in the lesson plans and the [full document](#) is included.

1.3 Curriculum Orientation

Check out this 1-hour [Orientation to the Curriculum](#) video

1.4 Python Versions (2 vs 3)

Python is an evolving language. Python 3 is a major upgrade to the language, released in 2010. There is a lot of existing software written under Python 2 and there is resistance to upgrading to Python 3 due to code breakage and cost. Just as a car part from a 10 year old model car will probably not fit a new model of the same car, Python 2 code probably would not run in a Python 3 environment. In Python 3 there are new features, significant upgrades "underneath" which makes the code run better and/or faster as well as no longer supporting (deprecating) some Python 2 capabilities. When looking at Python code, be careful to note whether it is Python 2 or Python 3.

1.4.1 This class will use Python 3

For those knowledgeable with Python 2, the following is a list of differences from Python 3 relevant to the 2nd semester intro course.

	Python 2	Python 3
Printing to console	<code>print 3.14</code>	<code>print (3.14)</code>
User input	<code>raw_input()/input()</code>	<code>input ()</code>
Integer arithmetic	<code>3/2 evaluates to 1</code>	<code>3/2 evaluates to 1.5</code>
Not equal to	<code><></code>	<code>!=</code>

1.4.2 IDE Selection

As with all software services, it is the school's sole decision to use the tool according to the use terms and privacy policies provided by its licensor and it is the school's responsibility to ensure the tool meets its IT policies.

1.5 Curriculum Issues

Please open an issue in GitHub if you encounter factual, spelling, or grammatical errors, sequencing problems (topics needed before they are taught), or incomplete/missing materials.

1.6 Giving feedback on the curriculum

TEALS intends for this curriculum to be a starting point for teachers. We'll continue to evolve, adapt the curriculum and associated materials. To participate in this process, we invite TEALS volunteers and classroom teachers using this curriculum to submit edits and suggestions via the [TEALS discussion forum](#) or in this [GitHub repository](#). If you'd like to suggest changes or additions to the curriculum, please submit a GitHub Pull Request containing your changes. As a best practice, each pull request should contain a singular atomic change.

1.7 Printing

The 2nd Semester Introduction to Computer Science can be printed by navigating to <https://aka.ms/TEALS2ndSemesterPDF>.

1.8 Creative Commons Attribution Non-Commercial Share-alike License

This curriculum is licensed under the [Creative Commons Attribution Non-Commercial Share-alike License](#), which means you may share and adapt this material for non-commercial uses as long as you attribute its original source, and retain these same licensing terms.

TEALS Program

[Home](#) | [Curriculum Map](#) | [Additional Readings](#) | [Change Log](#)

1 2nd Semester - Introduction to Computer Science Curriculum Map

1.1 Unit Order Considerations

As you implement this curriculum into your program, there are some things to consider when deciding what units best fits program goals.

- **Preparing students to take Advanced Placement Computer Science A** - It is recommended completing the units in the following order 1, 2, 3, 4, 7, 6, 8, 5(Optional)
- **Offering this course as a stand alone computer science course** - It is recommended completing the course units in the following order - 1, 2, 3, 4, 5, 6, 7, 8

1.2 Unit 1 Map - Introduction to Python

Lesson	Objectives	Lab	Days	Slide Deck
1.01: Set Up	Define and identify: IDE, Python. Identify the key concepts that will be covered in the course. Set up and log into an account for the course's online IDE. Save and turn in a file via the online IDE.	N/A	1	1.01
1.02: Interactive Mode	Define and identify: interpreter, string, integer, float, value, errors, console, expression. Use the Python interpreter to evaluate simple math expressions. Distinguish between an integer, float, and string.	Using the Interpreter	1	1.02
1.03: Script Mode and Variables	Define and identify: script, print, run, output, variable. Write a simple script and run it in the IDE. Print values out to the console (both composed values and from variables). Compare script mode vs interactive mode. Know how to store a value into a variable.	Printing & Variables	1	1.03
1.04: Variables Input	Define and identify: comments, storing, mutability, variable assignment, input. Assign and swap variables. Store user input into a variable.	Magic Genie	1	1.04
1.05: Quiz & Debugging	Demonstrate their understanding of key concepts covered up to this point. Define and identify: debugging, syntax errors. Analyze and respond to error messages.	N/A	1	1.05
1.06: Project	Apply basic Python knowledge about inputs/outputs and variables to create a game of Madlibs. Practice good debugging skills.	Mad Libs	2	1.06

1.3 Unit 2 Map - Data Types and Conditionals

Lesson	Objectives	Lab	Days	Slide Deck
2.01: Data Types & Casting	Define and identify: type, string, casting, floating point number (float), integer. Describe different representations of data in Python. Convert from one data type to another data type.	Casting	1	2.01
2.02: Booleans & Expressions	Define and identify: Boolean, expression, composition, True, False. Evaluate a Boolean expression. Compose Boolean expressions using and, or, not, <, >, and ==.	Can I or Can't I?	1	2.02
2.03: Conditionals	Define and identify: if, else, elif, conditionals, flow of control. Create chaining if statements. Understand how conditional statements alter the flow of control of a program.	Game Show	1	2.03
2.04: Lists	Define and identify: list, item, index, integer. Be able to access items from a list using the index. Create lists of different types. Use the length function.	College Chooser	1	2.04
2.05: Lists 2	Define and identify: slice, append, pop, remove. Slice a list. Add and remove elements from a list	Tic-Tac-Toe	1	2.05
2.06: Game Loop	Define and identify: while loop. Use a while loop to simulate game play.	Tic-Tac-Toe Revisited	1	2.06
2.07: Project	Use knowledge of lists, Booleans, conditionals, and while loops to create a text-based adventure game.	Text Monster Game	9	2.07

1.4 Unit 3 Map - Functions

Lesson	Objectives	Lab	Days	Slide Deck
3.01: Built In Functions	Define and identify: function, arguments, calling, importing, returning. Call the built-in randint function, using arguments. Utilize code other people have written in the Python documentation. Understand the difference between printing and returning.	Magic 8-Ball	1	3.01
3.02: User-Defined Functions	Define and identify: abstraction, def. Create functions.	Birthday Song & Random Cards	2	3.02
3.03: Return vs Print	Define and identify: return, none, void. Explain and demonstrate the difference between printing and returning.	War (Card Game)	1	3.03
3.04: Debugging and Scope	Define and identify: scope, aliasing, stack trace. Understand that changing a list in a function updates the list outside of the function. Understand that updating variables in a function does not affect the variable outside of the function. Understand global variables. Draw a simple stack trace.	Aliasing & Scope	1	3.04
3.05: Project	Use project planning skills to complete a longer-term project. Create functions to organize a project. Apply skills learned in units 1-3 to create a functioning program.	Oregon Trail	9	3.05

1.5 Unit 4 Map - Nested Loops and Lists

Lesson	Objectives	Lab	Days	Slide Deck
4.01: Looping Basics	Define and identify: for loop, item, iteration, scope. Recall looping in Snap! and reapply the concept in Python. Loop through (traverse) the items in a list. Be aware of the scope of variables during iteration.	de_vowel	1	4.01
4.02: For Loops	Define and identify: range. Use the range and len() function to update lists via for loops.	Getting Loopy	1	4.02
4.03: Nested For Loops	Define and identify: nested for loops, stack trace. Use nested for loops via a function and a for loop. Use nested for loops via two loops nested. Use a stack trace to understand and demonstrate the flow of nested for loops.	Nested For Loops	2	4.03
4.04: Nested Lists & Looping	Define and identify: nested list. Use nested for loops to traverse through nested lists.	Shopping List	2	4.04
4.05: Debugging and Quiz	Read and understand longer programs involving loops. Demonstrate knowledge of looping, lists, and nested loops/lists. Debug programs involving for loops and lists.	Debugging Practice	1	4.05
4.06: Project	Use project planning skills to complete a larger project. Utilize loops, lists, and nested loops/lists to create a Tic-Tac-Toe game.	Tic-Tac-Toe	9	4.06

1.6 Unit 5 Map (Optional) - Music Programming

Lesson	Objectives	Lab	Days	Slide Deck
5.01: Earsketch Intro	Define and identify: Digital Audio Workstation (DAW), sound tab, fitMedia(), setTempo(). Play beats using the above functions. Loop through items in a list. Be aware of the scope of variables during iteration.	Intro to EarSketch	1	5.01
5.02: EarSketch Music	Define and identify: rhythm, beat, tempo, measures, setEffect(), makeBeat(). Play beats using the functions. Loop through items in a list. Be aware of the scope of variables during iteration.	EarSketch Music	1	5.02
5.03: Earsketch Control Flow	Define and identify: modulo. Review looping and control structures. Use looping concepts in music making via EarSketch. Use control structures to create music.	Earsketch Control Flow	1	5.03
5.04: EarSketch User- Defined Functions	Define and identify: abstraction, section, A-B-A form. Create and apply user-defined functions to create songs with complicated form.	User-Defined Functions	1	5.04
5.05: Project	Create a complete song in EarSketch with multiple parts. Utilize EarSketch's features and functions.	EarSketch Song	5	5.05

1.7 Unit 6 Map - Dictionaries

Lesson	Objectives	Lab	Days	Slide Deck
6.01: Introduction to Dictionaries	Define and identify: dictionary, key, value. Create dictionaries of key-value pairs. Access and update items from dictionaries.	Dictionaries & Memes	1	6.01
6.02: Dictionaries Methods	Define and identify: pop, default value. Update values in a dictionary. Add values to a dictionary. Remove values from a dictionary.	Word Counter	1	6.02
6.03: Dictionaries of Lists	Create dictionaries with keys and values of different types. Update, append, or remove list values in a dictionary.	Dictionaries Storing Lists	1	6.03
6.04: Dictionaries Looping	Use loops to traverse through key/value pairs in a dictionary	Dictionaries Looping	1	6.04
6.05: Project	Use dictionaries to create the game Guess Who	Buying an Umbrella	7	6.05

1.8 Unit 7 Map - Introduction to Object Oriented Programming

Lesson	Objectives	Lab	Days	Slide Deck
7.01: User-Defined Types	Define and identify: class, instance, object, attributes. Create a class and instantiate. attributes to an instance. Manipulate instances and attributes through a function.	Create a Color Class	1	7.01
7.02: User-Defined Types, Part 2	Define and identify: self, __init__. Create a class with an __init__ method. Understand and use the self argument. Instantiate a class with arguments.	Pet Class	1	7.02
7.03: Methods	Define and identify: method, __str__, __add__, operator overloading. Create a class with an __init__ method. Understand and use the self argument. Instantiate a class with an argument.	Kangaroo Class	1	7.03
7.04: Inheritance	Define and identify: inheritance, parent class, child class. Create a class that inherits from another class. Overwrite methods of parent class in a child class.	Pokemon Child Classes	1	7.04
7.05: Project	Engage in class design before beginning coding. Apply what was learned with respect to classes, methods, and inheritance to create an implementation of Pokemon.	Pokemon	7	7.05

1.9 Unit 8 Map - Final Project

Lesson	Objectives	Lab	Days
8.01: Final Project Brainstorming and Evaluating	Recall project planning basics from last semester. Identify factors to use when choosing between project ideas. Rank a group of proposed project ideas using the identified factors.	N/A	1
8.02: Defining Requirements	Define key scenarios for a project and the features required to implement each scenario. Explain the importance of wireframing when designing an application.	N/A	1
8.03: Building a Plan	Identify the main components of a functional project specification and explain the purpose of each section. Develop a project idea into a full, detailed specification.	N/A	1
8.04: Project Implementation	Use the skills developed throughout the course to implement a medium- to large-scale software project. Realistically evaluate progress during software development and identify when cuts are necessary. Prioritize features and scenarios and choose which should be eliminated or modified if/when resources and/or time become limited.	N/A	15

1.10 Supplemental Culture Day Lessons

Lesson	Objectives	Lab
01: Binary Day	Define and identify: binary. Describe different representations of data. Represent decimal numbers in binary.	N/A

formatted by Markdeep 1.093 

TEALS Program

[Home](#) | [Curriculum Map](#) | [Additional Readings](#) | [Change Log](#)

1 Unit 1 - Text-based vs. Block-based Programming

This unit introduces students to a text based coding IDE, Console and starts to transition them away from a block based language and into Python.

1.1 Essential Questions

- What is Python?
- What is an IDE?
- What are the basics to programming with Python?
- How do you use the console?
- How do apply what I learned in Snap to programming in Python?

1.2 Pacing Guide

1.2.1 Timing

1 Day = 50 minute class period

Lesson	Days	Slide Decks
1.01: Set Up	1	1.01 Slide Deck
1.02: Interactive Mode	1	1.02 Slide Deck
1.03: Script Mode and Variables	1	1.03 Slide Deck
1.04: Variables Input	1	1.04 Slide Deck
1.05: Quiz & Debugging	1	1.05 Slide Deck
1.06: MadLibs	2	1.06 Slide Deck
Total Days	7	
Total Minutes	350	

1.3 Key Terms

- IDE
- Python
- Intrepeter
- String
- Integer
- Float
- Value
- Errors
- console
- Expression
- Script
- Print
- Run
- output
- Variable
- Script Mode
- Interactive Mode
- Comments
- Storing
- Mutability
- Variable Assignment
- Input
- Debugging
- Syntax

formatted by [Markdeep 1.093](#) 

TEALS Program

[Home](#) | [Curriculum Map](#) | [Additional Readings](#) | [Change Log](#)

1 Lesson 1.01: Set Up

1.1 Learning Objectives

Students will be able to...

- Define and identify: **IDE, Python**
- Identify the key concepts that will be covered in the course
- Save and turn in a file

1.2 Materials/Preparation

- Update the collaboration policy, IDE information, grading percentages, and general syllabus based on your preferences and your school's requirements
- With your teaching team determine which IDE to use with your class.
- **Python Code:** code to display
- Student Notebooks
- Decide which IDE will be used for the classroom for the semester
- Note: As with all software services, it is the school's sole decision to use the tool according to the use terms and privacy policies provided by its licensor and it is the school's responsibility to ensure the tool meets its IT policies.

1.3 Pacing Guide

Duration	Description
5 Minutes	Do Now
15 Minutes	Syllabus
10 Minutes	Python/IDE Intro
25 Minutes	Submit Work

1.4 Instructor's Notes

1.4.1 1. Do Now

- Students download and install IDE to be used for class this semester.

1.4.2 2. Syllabus

- Guide students through the syllabus, pausing for questions as needed.
- Make sure to note the overarching goals for the course: create functioning Python programs, learn to use basic data types, and learn to use a text-based language.
- Present demo of a program written in Python to hook students and to give them an idea of what they will be working with.

1.4.3 3. Python/IDE Intro

1.4.3.1 Introduce Python

- **Python** is a text-based programming language, uses tabbing/indentation to control execution, and is known for its readability.
- Show [Python Code]. Ask students to write down at least 3 specific things they observe about the program/code in their notebooks.
- Possible responses might include: indentation, different colors of text, using “print” instead of “say” (Snap).
- Discuss what students observed.

1.4.3.2 Introduce IDE - Integrated Development Environment

- An **IDE** is an application that allows you to create, edit, save, and run programs.
- Note that with Python, we can't use drag and drop blocks anymore - it is a language that must be typed into an IDE.
- Talk to students about the different areas and parts of the IDE, use the [Python Code](#) to demo once more how it looks/works.

1.4.3.3 Activity (instructor should demonstrate while students are following along on their computers)

- Hello World exercise
- Display [Python Code].
- Have students create a new Python file
- Save file as “hello_world.py”
- Have students type in the displayed code into the blank hello_world.py file
- If students finish early, encourage them to explore the interface and to try creating a program that prints their name.

1.5 Forum discussion

Lesson 0.1: The First Day ([TEALS Discourse Account Required](#))

TEALS Program

[Home](#) | [Curriculum Map](#) | [Additional Readings](#) | [Change Log](#)

1 Lesson 1.02: Interactive Mode

1.1 Learning Objectives

Students will be able to...

- Define and identify: **interpreter, string, integer, float, value, errors, console, expression**
- Use the Python interpreter to evaluate simple math expressions
- Distinguish between an integer, float, and string

1.2 Materials/Preparation

- [Lab - Using the Interpreter \(printable lab document\)](#) ([editable lab document](#))
- [Associated Reading - section 1.1](#)
- Ensure all students are able to log into the system
- Go through the lab so that you are familiar with the requirements/results and can assist students

1.3 Pacing Guide

Duration	Description
5 Minutes	Do Now
10 Minutes	Lesson
15 Minutes	Lab Part 1/Discussion
20 Minutes	Lab Part 2/Discussion
5 Minutes	Debrief

1.4 Instructor's Notes

1.4.1 1. Do Now

- Display the online IDE and asks students to identify the key parts (menu bar items [run, console, +], code panel, console panel) of the IDE.

- After going over the three parts of the IDE have students check that they can still log into their IDE account.

1.4.2 2. Lesson

1.4.2.1 Guided Activity

- Have students all bring up their console on their computer
- The part on right half of the screen is called a **console**:
- The **console** is a place where you can interact with a program.
- The **interpreter** runs Python code.
- To run the Python interpreter, type code into the console and hit “Enter” or click “Run”, the code executes immediately.
- Make sure all the students are able to do this and then give out the lab worksheet.

1.4.3 3. Lab Part 1

Give students time to work on section 1,

1.4.3.1 Discussion Section 1

- What does the // do? How is that different from /? And how are those different from %?
- What's the difference between 3.0 and 3?
- Go over the following two terms
 1. **Floats**: a data type, number with a decimal point.
 2. **Integers**: a data type, number without a decimal point.
- Now, give students time to work on section 2

1.4.3.2 Discussion Section 2

- What happened when you typed in a?
- What do you think that error message mean?
- Go over **String**: a data type, characters surrounded in single or double quotes.
- Now, give students time to work on section 3

1.4.3.3 Discussion Section 3

- What was the difference between the two inputs?
- Strings can be combined using +.
- What do you think the error message means?
- You can't combine different types!
- Now, give students time to work on section 4

1.4.3.4 Discussion Section 4

- What error did you get? What do you think that means?
- What happens when you multiply strings?

1.4.4 4. Lab Part 2

- Define **expression**: a combination of values and operators (and variables)
- Ask students to give an example of an expression
- Make sure students write down their predictions before going to the interpreter/IDE to check the actual output.

1.4.5 5. Debrief

- Discuss any surprising/unexpected results.
- Remind students of adding strings together using +.
- Talk about how single and double quotes are interchangeable
- Multiplying strings
- Order of Operations is the same as what students have learned in math class.
- Discuss why it might be helpful to have an interactive console. How is it different than snap?

1.4.6 Accommodation/Differentiation

If students are moving quickly, have students practice higher order-of-operations problems. You can also have them practice assigning values to variables.

1.5 Forum discussion

[Lesson 1.02: Interactive Mode \(TEALS Forums Account Required\)](#)

formatted by [Markdeep 1.093](#) 

TEALS Program

[Home](#) | [Curriculum Map](#) | [Additional Readings](#) | [Change Log](#)

1 Lab 1.02 - Using the Interpreter

1.1 Part 1

Using the interpreter, type in the expressions below. Copy and paste the output into the output column. If the result is unexpected, note that in the third column.

1.1.1 Section 1

	Input	Output	Did it do something unexpected?
a	$5 + 2 * 2$		
b	$2/3$		
c	$2.0 * 1.5$		
d	$(2 + 3) * 10$		
e	$5.0 // 2$		
f	$5.0 \% 2$		

1.1.2 Section 2

	Input	Output	Did it do something unexpected?
a	a		
b	'a'		

1.1.3 Section 3

	Input	Output	Did it do something unexpected?
a	'a + b'		
b	'a' + 'b'		

1.1.4 Section 4

	Input	Output	Did it do something unexpected?
a	'a' * 'b'		
b	'a' * 2		

1.2 Part 2

1.2.1 Before going to the IDE

1. For each item, predict the data type of the result and enter into the "String/Integer/Float" column.
2. Next, predict the value of the result for each item and enter into it into the "Prediction of Result" column.

	Expression	String/Integer/Float	Prediction of Result	Interpreter Result
a	<code>10 * 2</code>	integer	20	20
b	<code>.5 * 2</code>			
c	<code>10/2</code>			
d	<code>10%2</code>			
e	<code>2 ** 3</code>			
f	<code>(2+5)*3</code>			
g	<code>2 + 5 * 3</code>			
h	<code>'ab' + '12' + '3'</code>			
i	<code>x</code>			
j	<code>"ab" + "cd"</code>			
k	<code>'abc' * 2</code>			
l	<code>'1'*2 + '2' * 3</code>			
m	<code>1 * 2 + '3' * 2</code>			
n	<code>'A' ** 2</code>			
o	<code>'bc' % 2</code>			
p	<code>'bc' / 2</code>			

1.3 Now go to the IDE

Use the interpreter to evaluate the expressions, write down results in the "Interpreter Result" column.

TEALS Program

[Home](#) | [Curriculum Map](#) | [Additional Readings](#) | [Change Log](#)

1 Do Now 1.03

Open and save a new project called `DoNow103` in the IDE. Run a program by clicking the ">" Run button. See the instructions below for how to open, save, and run a file in the online IDE.

Practice typing the following expressions in the editor and running the code.

1.1 Expression 1

```
2 * 3 * 5
```

1.1.1 Expression 2

```
"abc"
```

1.1.2 Expression 3

```
"abc" + "bde"
```

1.2 Now try typing the statement below into the file. Save and run the code

```
print(2*3*5) //
```

Write down the result. Explain what `print` does. Try printing out 3 different values.

1.3 SNAP Flashback – Print Command

Snap! Print Command

1.4 How to open, save, and run a file

1. Open a Python 3 project:

Open a project

2. Repl.it autosaves projects
3. Run the program

Run Python 3

formatted by Markdeep 1.093 

TEALS Program

[Home](#) | [Curriculum Map](#) | [Additional Readings](#) | [Change Log](#)

1 Lab 1.03 - Printing & Variables

1.1 Part 1 - Printing Practice

Practice typing out some statements in the editor part of the IDE, then hit “Run” at the top of the screen:

Expression	Expected Output	Did anything unexpected happen?
<code>print("1")</code>		
<code>print(1)</code>		
<code>print(1 + 2)</code>		
<code>print("1" + "2")</code>		
<code>print("this" + " " + "is" + " " + "a" + " " + "sentence" + ".")</code>		

1.2 SNAP Flashback - Print Comparison

Code

Image

1.3 Part 2 - Variables Practice

1.3.1 1. In your Console

1.3.1.1 Type and run the following

```
animal = "dogs"
print(animal + " are really cool.")
```

//

1.3.2 In your Notebook

1.3.2.1 Respond to the following

1. What happens?
2. How would you make the program print out "cats are really cool" instead?

1.3.3 2. In your Console

1.3.3.1 Type and run the following code

```
print(dogs + " are cool.")
```

//

1.3.4 Continue In your Notebook

1.3.4.1 Respond to the following questions

1. What output does this produce?
2. Why does this happen?

1.3.5 3. In your Console

1.3.5.1 Rewrite the following Snap! Program in Python

snap_blocks_variables

1.4 Part 3 - Four Fours

1.4.1 The four fours challenge

Using four 4's and any operations, try to write equations that have the numbers from 0 to 4 as the answer. You should use Python's arithmetic operations:

- \+ addition
- \- subtraction or negation
- * multiplication
- / division
- () parentheses for grouping
- ** power

You may also use 44 or 4.4, which count as two fours, or .4, which counts as one four. For example, one solution for zero is:

```
print("Zero is", 44-44)
```

//

Can you find a different solution?

Here are what the results, but not the source code, will look like. (Note: answers may have trailing zeros if floating point arithmetic is used which is fine, i.e. 1 may be displayed as 1.0)

```
Zero is 0
One is 1
Two is 2
Three is 3
Four is 4
```



1.5 Bonus

Print the output below, but only using **one** line of code. Feel free to use online resources.

```
Wow!
This is on a new line!
```



1.6 Bonus 2

Can you find four fours for 5 to 10?

formatted by [Markdeep 1.093](#) 

TEALS Program

[Home](#) | [Curriculum Map](#) | [Additional Readings](#) | [Change Log](#)

1 Do Now 1.04

1.1 1. In your Console

1.1.1 Type and run the following code

```
a = input("What is your name? ")
# a = "cats and dogs"
# meow
print("Hello there, " + a)
```



1.2 In your Notebook

1.2.1 Answer the following questions

1. Read through the code and write down what you expect the printed results to be?
2. Run the code and write down the actual printed result?
3. Briefly describe what the # does?
4. Briefly describe what `input` does?

1.3 Snap to Python

Convert the following to Python code:

Snap Input

1.4 Swapping

1.4.1 2. In your Console

1.4.1.1 Type and run the following

```
a = "this sentence should go second"
b = "this sentence should go first."
# your code starts here
```

```
# your code ends here
print(a)
print(b)
```

- //
1. Run the program. What is the output?
 2. Write code to swap the values of variables a and b.

formatted by Markdeep 1.093 📝

TEALS Program

[Home](#) | [Curriculum Map](#) | [Additional Readings](#) | [Change Log](#)

1 Lab 1.04 - Magic Genie

1.1 Project Objective

- Use Python to interact with variables and user input
- Create a genie program. Save the file as `magic_genie.py`.

1.1.1 Project Specifications

- Have the program introduce itself
- Have the program ask for three separate wishes
- Print all the wishes together

magic_genie_output

- There are some repeated strings in this genie program. Move those into variables

1.1.2 Genie Confusion

Now it's time to make your genie confused. Edit your code to have him print your first wish as your last wish, and your second wish as your first wish, and your third wish as your second wish.

magic_genie_output_confused

1.2 SNAP Flashback – Magic Genie

Genie Code - Snap

1.2.1 Hint

Remember to add spaces you can combine " " to the end of your string using the + operator. So `print("hello" + " " + "student")` would print hello student

TEALS Program

[Home](#) | [Curriculum Map](#) | [Additional Readings](#) | [Change Log](#)

1 Lesson Plan 1.05: Quiz & Debugging

1.1 Learning Objectives

Students will be able to...

- Demonstrate their understanding of key concepts covered up to this point
- Define and identify: **debugging, syntax errors**
- Analyze and respond to error messages

1.2 Materials/Preparation

- Download quiz and Answer key(access protected resources by clicking on "Additional Curriculum Materials" on the [TEALS Dashboard](#))
- [Associated Readings 1.3](#)
- Take the quiz
- After taking the quiz yourself, take a look at the answer key provided in the additional curriculum materials.
- Create a scoring rubric
- Look through <http://tinyurl.com/TEALS-Python-Errors> web tutorial to dig deeper yourself into analyzing and responding to errors

1.3 Pacing Guide

Duration	Description
5 Minutes	Welcome and Review
25 Minutes	Quiz
25 Minutes	Debugging Activity

1.4 Instructor's Notes

1.4.1 1. Welcome and Review

- Ask students for any final questions before passing out the quiz.

1.4.2 2. Quiz

1.4.3 3. Debugging Activity

- Explain to students that you will now be exploring how to read analyze and respond to errors in code.
- Ask students to read through it and predict what will be printed out.

```
favorite_number_str = input("What is your favorite number: ")  
birth_month_str = input("What month where you born in: ")  
lucky_number = int(favorite_number_str) + int(birth_month_str)  
print("Your lucky number is " + lucky_number)
```



- Remind students that when reading through code we go line by line, as if we are the interpreter.

1.4.3.1 Demonstration

- Run the code, display the stack trace, and have students analyze the error message reported.
- Explain that, much like in Snap, **debugging** is the process of tracking and fixing errors in your code.
- Indentation Errors: Errors the students are most likely to have seen
- “IndentationError: unindent does not match any outer indentation level”
- Ask student why these errors are caused and how they find/fix this type of error? Suggest using the tab key to indent and the shift-tab to remove an indent as ways to avoid the error.
- Direct students to work through and complete this web tutorial on [debugging](#)

1.4.4 Accommodation/Differentiation

Make sure to provide extended time on the quiz for any students who need extra time

1.5 Forum discussion

[Lesson 1.05: Quiz and Debugging \(TEALS Forums Account Required\)](#)

formatted by [Markdeep 1.093](#) A small icon of a pencil with a horizontal stroke through it, indicating it's a link to the original document or a specific edit.

TEALS Program

[Home](#) | [Curriculum Map](#) | [Additional Readings](#) | [Change Log](#)

1 Lesson 1.06: Project

1.1 Learning Objectives

- Apply basic Python knowledge about inputs/outputs and variables to create a game of Madlibs
- Practice good debugging skills

1.2 Materials

- Project Spec - Mad Libs ([Printable Project Spec](#)) ([Editable Project Spec](#))
- Alternate Project Spec - Magic Square ([Printable Alternate Project Spec](#)) ([Editable Alternate Project Spec](#))
- Solution (access protected resources by clicking on "Additional Curriculum Materials" on the [TEALS Dashboard](#))
- [Editable Grading Rubric](#)

1.3 Preparation

- Practice running the example code
- Read through the project specifications so that you can completely communicate the requirements of the project
- Review [4 Steps to Solve Any CS Problem]

1.4 Day 1 Pacing

Duration	Description
5 Minutes	Quiz Debrief
10 Minutes	Project Overview
40 Minutes	Project Work

1.5 Day 2 Pacing

Duration	Description
45 Minutes	Project Work
10 Minutes	Wrap Up - Student Demos

1.6 Instructors Notes

1.6.1 1. 4 Steps to Solve Any CS Problem

- Introduce students to the [4 Steps to Solve Any CS Problem]

1.6.2 2. Project Overview

- Introduce students to the Mad Lib concept by using a short, written out Mad Lib on the whiteboard, poster paper, or projector.
- Pass out and the project specification and walk students through all the requirements and potential challenges.
- Emphasize that prompts must ask for the correct noun-verb combinations.
- Encourage students to look at the grading rubric on page two repeatedly throughout the project to ensure they are meeting all the requirements.
- Demo a sample project solution (access protected resources by clicking on “Additional Curriculum Materials” on the [TEALS Dashboard](#)) for students to see how a completed program should function.
- Identify the sub problems of Mad Libs
- Have students list what variables, inputs, and print statements they will need

1.6.3 3. Project Work

- This project is a summative assessment for the unit. Students should be demonstrating mastery of all the skills covered.
- Most students will require roughly 1 hour of total work time to complete the project
- Assess the progress of your students regularly using such techniques as asking them to demonstrate their incomplete programs, tracking questions asked during lab time, and/or utilizing peer reviews.
- Adjust the amount of time allowed for the project to fit the needs of your students
- It is vital that nearly all students complete the project before moving on
- If most students have the ability to work on assignments at home, the amount of in-class time provided can be reduced if necessary.
- If this approach is taken, be sure to make accommodations for students who not able to work at home, such as after school lab hours
- Ensure that students are able to ask questions in class throughout the project

1.6.4 4. Wrap Up - Student Demos

- Celebrate and showcase student work once projects are completed.
- Have students demonstrate their Mad Libs for the class, with the class choosing what nouns/verbs/etc. to use for the story.

1.7 SNAP Flashback - MadLibs

1.8 Accommodation/Differentiation

Ask students to research casting. Have them add, subtract, or multiply values as part of the story.

1.9 Grading

1.9.1 Objective Scoring Breakdown

[Editable Grading Rubric](#)

Points	Percentage	Objective	Unit Location
2	10%	Students can correctly use the IDE	1.01
6	28%	Student can correctly identify and store variable types	1.02 1.04
3	14%	Student can use the print function	1.03
5	24%	Student can decompose a problem to create a program from a brief	
5	24%	Student uses naming/ syntax conventions and comments to increase readability	
21	Total points		

1.9.2 Scoring Consideration

You may need to adjust the points in order to fit your class. Treat the percentages as a guide to determine how to weight the objectives being assessed.

1.10 Forum discussion

[Lesson 1.06: Mad Libs \(TEALS Forums Account Required\)](#)

TEALS Program

[Home](#) | [Curriculum Map](#) | [Additional Readings](#) | [Change Log](#)

1 Project 1: Mad Libs

Using Python, students will use variables, input, and printing to create a Mad Lib. You will also practice *designing* a project by planning out your Mad Lib before implementing it. Part of the project is to use your creativity to design your own unique story.

1.1 Overview

Mad Libs are a fun way to tell a story. The story is pre-written except for a few missing words. The story is hidden from the user. The user is asked a series of questions in order to fill in the missing words before seeing the story. Then the story is read off with the user's words mixed in!

1.2 Details

1.2.1 Behavior

- The program will print out the title of the Mad Libs story, as well as a short explanation of game play

A Day `in` NYC: A Mad Lib.
Welcome! You are about to play a fantastic word game.
I will ask you `for` nouns, verbs, adjectives, proper nouns `and` adverbs.
Using those words I will create an unexpected story `for` you! //
- The program should then prompt the user to enter in nouns, verbs, adjectives, proper nouns, and adverbs

Day `in` New York City: A Mad Lib
Instructions. The program will prompt `for` a type of word to enter. After all words are entered the program will `print` a story
Enter a proper noun: Ariana Grande
Enter a place: The Standard
Enter another place: Duane Reade
Enter an adverb: quickly
Enter a noun: donut
Enter an adjective: slimy
Enter an adverb: foolishly
Enter a verb: prance
Enter a place: Times Square
Enter an adjective: beautiful
It`was a beautiful day in New York City`
Our`hero Ariana Grandewas on a walk from the Standard`
Suddenly`slimy donut appeared out of nowhere!!!`



- After all the words have been entered. The program will print out the story. You will need to create a story of your own choosing. Keep it clean and fun. Here is an example of a day in New York City.

A Day **in** NYC: It was a beautiful day **in** New York City. Our hero Ariana Grande was on a walk **from** the Standard to Duane Reade. Ariana Grande was walking rather quickly because he/she had lived **in** New York **for** a few months. Suddenly, a slimy donut appeared out of nowhere!!! Ariana Grande decided to prance foolishly instead of dealing **with** the situation. Thrown off **from** Duane Reade, Ariana Grande decides to go to Times Square instead. What a // beautiful day **in** New York.

1.2.2 Implementation Details

Plan out your story on pencil and paper first, before you start implementing the program.

1. Create your story
2. Select the missing words
3. Determine each words part of speech
4. Create introduction
5. Create questions
6. Divide story into print statements

As mentioned above the program must request words from the user. The following **must** be included in the program:

- 10 different words inputted
- Variable names should correspond to the part of speech requested and part of the story they belong to (e.g. noun1, verb2, etc.)
- You may only use 3 print statements to tell your story

formatted by Markdeep 1.093 📝

TEALS Program

[Home](#) | [Curriculum Map](#) | [Additional Readings](#) | [Change Log](#)

1 Alternate Project 1: Magic Square

Created by Brian Weinfeld

Using Python, you will use variables, input, and casting to create a Magic Square.

1.1 Overview

Pick a number from 21-65. **42**, you say? OK! Check this out!

```
22 01 12 07  
11 08 21 02  
05 10 03 24  
04 23 06 09
```



If you add up all the numbers in each row, they total 42. ($22 + 1 + 12 + 7 = 11 + 8 + 21 + 2 = 42$)

If you add up all the numbers in each column, they total 42. ($22 + 11 + 5 + 4 = 1 + 8 + 10 + 23 = 42$)

If you add up all the numbers in each diagonal, they total 42. ($22 + 8 + 3 + 9 = 7 + 21 + 10 + 4 = 42$)

It is the same for each of the four corners, and each 2×2 block as well. ($22 + 4 + 9 + 7 = 42$)

This is called a **Magic Square** and for this project, you are going to create a program that lets users select a number and create a magic square from that number.

1.2 Details

1.2.1 Behavior

```
Welcome to Magic Square  
Enter a number from 21 to 65: 42  
You have entered 42
```

Here **is** your Magic Square:

```
22 01 12 07  
11 08 21 02  
05 10 03 24  
04 23 06 09
```



1.2.2 Implementation Details

Believe it or not, Magic Squares are not difficult to make! Watch the following video to see how to make a Magic Square for any given number:

1.2.3 Challenge

This section contains additional components you can add to the project. These should only be attempted **after** the project has been completed.

- What happens if the user enters a number outside the range of 21-65? Try to check for this and print an error message!
- What happens if the user doesn't enter a number at all and enters a word instead? Try to check for this and print an error message!
- Build a Magic Square with a small number like 22. The Magic Square isn't aligned properly and hard to read. Try to fix this!

formatted by [Markdeep 1.093](#) 

TEALS Program

[Home](#) | [Curriculum Map](#) | [Additional Readings](#) | [Change Log](#)

1 Unit 2 - Data Types, Arguments, Lists and Loops

1.1 Essential Questions

- How do you apply Data Types in Python?
- How do you apply Arguments in Python?
- How do you apply lists in Python?
- How do you apply a While Loops in Python?

1.2 Pacing Guide

1.2.1 Timing

1 Day = 50 minute class period

Lesson	Days	Resources
2.01: Data Types & Casting	1	2.01 Slide Deck
2.02: Booleans & Expressions	1	2.02 Slide Deck
2.03: Conditionals	1	2.03 Slide Deck
2.04: Lists	1	2.04 Slide Deck
2.05: Lists 2	1	2.05 Slide Deck
2.06: Game Loop	1	2.06 Slide Deck
2.07: Text Game	9	2.07 Slide Deck
Total Days	15	
Total Minutes	750	

1.3 Key Terms

- type

- string
- casting
- Float
- Integer
- Boolean
- Expression
- Composition
- True
- False
- If
- Else
- Elif
- Conditionals
- Flow of control
- List
- Item
- Index
- Slice
- Append
- Pop
- Remove
- While Loop

formatted by [Markdeep 1.093](#) 🎨

TEALS Program

[Home](#) | [Curriculum Map](#) | [Additional Readings](#) | [Change Log](#)

1 Lesson 2.01: Data Types & Casting

1.1 Learning Objectives

Students will be able to...

- Define and identify: **type, string, casting, floating point number (float), integer**
- Describe different representations of data in Python
- Convert from one data type to another data type

1.2 Materials/Preparation

- [Do Now](#)
- [Lab - Casting \(printable lab document\)](#) ([editable lab document](#))
- [Associated Readings 2.1](#)
- Read through the do now, lesson, and lab so that you are familiar with the requirements and can assist students

1.3 Pacing Guide

Duration	Description
5 Minutes	Do Now
10 Minutes	Lesson
35 Minutes	Lab
5 Minutes	Debrief

1.4 Instructor's Notes

1.4.1 1. Do Now

- Project the Do Now on the board, circulate around the class to check that students are working and understand the instructions.

1.4.2 2. Lesson

1.4.2.1 Discussion

- Ask students to define **type**. Talk about types as a way to represent data (give examples of strings, int, and float).
- Ask students what they thought typing the command `str(123)` does.

1.4.2.2 Instruction

- Define this process of changing data types as **casting**.
- Define the `int` function if the students were unable to guess it from the do now.
- Take a few minutes to have students write down how they would produce the following output:

```
```python > Give me a number you want to multiply by 2: 4 8 => None ```
```
- Explain to students that in Python
  - When asking for input from the user the input is automatically stored as a string and will
  - if the input will need to be used for calculations or another purpose it will need to be casted to another data type.

### 1.4.2.3 Student Sharing

- Call on 2-3 students to write their answers on the board.
- Discuss what would happen if the user types in 1.5 instead of 4.
- If input is a float, can cast with `float(num)`
- `type`: ask students what they think `type('a')` would output.
- Why might you want to use `type`?

## 1.4.3 3. Lab

- Practice predicting what casting will do to inputs.
- Create a halving program and a program that converts halves to whole numbers.

## 1.4.4 4. Debrief

- Check student progress and completion of the lab, wrap up by taking any final questions.

## 1.5 Accommodation/Differentiation

---

If students are moving quickly, it is possible to introduce the concepts of Booleans here. Discuss how students would represent binary (0's and 1's). In practice these translate to True and False. Convert numbers to Boolean, and Booleans to numbers.

## 1.6 Forum discussion

---



# TEALS Program

[Home](#) | [Curriculum Map](#) | [Additional Readings](#) | [Change Log](#)

## **1 Do Now 2.01**

---

### **1.1 Open up the console**

---

#### **1.1.1 Type the following code**

```
x_stage1 = 123
y_stage1 = 456

x_stage2 = str(x_stage1)
y_stage2 = str(y_stage1)

print(x_stage2 + y_stage2)
```

//

### **1.2 In your Notebook**

---

#### **1.2.1 Answer the following**

1. What type are the variables `x_stage1` and `y_stage1`?
2. What type are the variables `x_stage2` and `y_stage2`?
3. How would you convert a string, '100' to an integer?

formatted by [Markdeep 1.093](#) ↗

# TEALS Program

[Home](#) | [Curriculum Map](#) | [Additional Readings](#) | [Change Log](#)

## 1 Lab 2.01 - Casting

---

### 1.1 In your notebook

---

#### 1.1.1 Predict what the following inputs will result in

Once you have filled in the “prediction” column, check your answers in interactive mode and write the actual result.

Input	Prediction	Result
<code>float('1')</code>		
<code>str(1 + '2')</code>		
<code>str('2')</code>		
<code>int('abc')</code>		
<code>int(float('1.6'))</code>		
<code>float(int(1.6))</code>		
<code>str(float(1))</code>		

### 1.2 In your Console

---

1. Create a program which will take in an input and print out that input divided by 2.
2. Alter one line of that program to return only whole numbers.

#### 1.2.1 Bonus

Make your program have two modes: an integer mode and a float mode. Add another input to ask which mode the user wants to use. If the user is in integer mode print out integers, otherwise print out float. Feel free to research Python docs (This is a concept covered in Snap)

# TEALS Program

[Home](#) | [Curriculum Map](#) | [Additional Readings](#) | [Change Log](#)

## 1 Lesson 2.02: Booleans & Expressions

---

### 1.1 Learning Objectives

---

Students will be able to...

- Define and identify: **Boolean, expression, composition, True, False**
- Evaluate a Boolean expression
- Compose Boolean expressions using and, or, not, <, >, and ==

### 1.2 Materials/Preparation

---

- [Do Now](#)
- [Lab - Can I or Can't I? \(printable lab document\)](#) ([editable lab document](#))
- [Associated Readings 2.2](#)
- Read through the do now, lesson, and lab so that you are familiar with the requirements and can assist students.

### 1.3 Pacing Guide

---

Duration	Description
5 Minutes	Do Now
10 Minutes	Lesson
35 Minutes	Lab
5 Minutes	Debrief

### 1.4 Instructor's Notes

---

#### 1.4.1 1. Do Now

- Project the Do Now on the board
- Circulate around the class to check that students are working and understand the instructions

## 1.4.2 2. Lesson

### 1.4.2.1 Discussion

- After 5 minutes of students working on the Do Now, ask students to recall what a Boolean is and how they used Booleans in Snap!
- Ask students what values they saw in part 1 of the Do Now (answer should be True or False)

### 1.4.2.2 Instruction

- **Boolean expression:** is an expression that evaluates to either true or false.
- Ask Students about the difference between = and ==.
- = is for assignment of value
- == builds a Boolean expression and is a way to compare two values
- Remind students of Boolean expressions in Snap!

*Snap Boolean Expressions*

*Snap Boolean Expressions*

- Ask the students to recall what and, or and not did.
- Give students additional time to finish completing part 2 of the Do Now, if needed.
- Have a student write up the expression they used to update the can\_get\_license code.
- Discuss with students part 3 of the Do Now and how 'or' is used two different ways.

### 1.4.2.3 Poll students -

- how many Boolean expressions are used?
  - Answers here may vary depending on the students' code.

### 1.4.2.4 Instruction Part 2

- Define **composition:** Using an expression as part of a larger expression, or a statement as part of a larger statement. You can use parentheses to compose expressions as well.
- Parentheses: In Snap! to compose many expressions they were nested together by simply putting blocks one after another. However, in Python if you want certain things to be evaluated together, use parentheses.

## 1.4.3 3. Lab

- Evaluate expressions with and, or, and not
- Given written out rules, students will convert them into Boolean expressions.
- Create a large expression using variables that describes you.

## 1.4.4 4. Debrief

- Check student progress and completion of the lab, wrap up by taking any final questions.

## **1.5 Accommodation/Differentiation**

---

If students are moving quickly, use this opportunity to go over truth tables (or ask them to research De Morgan's Law)

### **1.5.1 Convert the following SNAP Truth Table program into Python**

*Snap Truth Tables*

## **1.6 Forum discussion**

---

[Lesson 2.02: Booleans & Expressions \(TEALS Discourse Account Required\)](#)

formatted by [Markdeep 1.093](#) 



# TEALS Program

[Home](#) | [Curriculum Map](#) | [Additional Readings](#) | [Change Log](#)

## 1 Lab 2.02 - Can I or Can't I

---

### 1.1 In your Notebook

---

Predict if each of the following examples will produce a True or False output. Check your answers in interactive mode.

#### 1.1.1 Example 1

```
>>> a = 100
>>> b = "science"
>>> a > 75 and b == "science"
```



#### 1.1.2 Example 2

```
>>> a = 100
>>> b = "science"
>>> a > 75 and b != "science"
```



#### 1.1.3 Example 3

```
>>> a = 100
>>> b = "science"
>>> a > 75 or b != "science"
```



#### 1.1.4 Example 4

```
>>> a = 100
>>> b = "science"
>>> c = True
>>> not c and a > 75 and b == "science"
```



### 1.2 In your Console

---

#### 1.2.1 Complete the following coding challenge

1. Create a “Can I be President?” program, which determines if the user meets the minimum requirements for becoming the President of the United States. Have the user input the information needed.

**The minimum requirements to be president of the United States are:**

- Older than 35
  - Resident of US for 14 Years
  - Natural born citizen
  - Print True if the person could be president and False if they can't be president.
2. Create a "I can't be President?" program. Print True if the user cannot be President and False if they can be President.
3. Create a "Can I ride the roller coaster?" program. A roller coaster has the rule that a rider has to be over the height of 50 inches. Because of a legal loophole, if you are over the age of 18 you can ride regardless of your height. If you are allowed to ride, the coaster costs 4 quarters (although the operator accepts tips so more money is appreciated).
- Also, the theme park sells frequent rider passes: with a frequent rider pass the roller coaster costs only 2 quarters. Ask the user how tall they are in inches, their age, how many quarters they have, and if they have a frequent rider pass. Print True if the person can ride and False if they can't.

## 1.3 Bonus

---

### 1.3.1 Are the following expressions equivalent? Research DeMorgan's Laws and write why you think they are the same or why they are not the same

```
not(x or y) == not x and not y
```

```
not(x and y) == not x or not y
```

formatted by [Markdeep 1.093](#) 

# TEALS Program

[Home](#) | [Curriculum Map](#) | [Additional Readings](#) | [Change Log](#)

## 1 Lesson 2.03: Conditionals

---

### 1.1 Learning Objectives

---

Students will be able to...

- Define and identify: if, else, elif, conditionals, flow of control
- Create chaining if statements
- Understand how conditional statements alter the flow of control of a program

### 1.2 Materials/Preparation

---

- [Do Now](#)
- [Lab - Game Show \(printable lab document\)](#) ([editable lab document](#))
- Solution (access protected resources by clicking on "Additional Curriculum Materials" on the [TEALS Dashboard](#))
- [Associated Readings 2.3](#)
- Read through the do now, lesson, and lab so that you are familiar with the requirements and can assist students

### 1.3 Pacing Guide

---

Duration	Description
5 Minutes	Do Now
10 Minutes	Lesson
35 Minutes	Lab
5 Minutes	Debrief

### 1.4 Instructor's Notes

---

#### 1.4.1 1. Do Now

- Project the Do Now on the board, circulate around the class to check that students are working and understand the instructions.

- Students should quickly realize that they do not have all the tools necessary to complete the task.

## 1.4.2 2. Lesson

### 1.4.2.1 Instruction

- Explain that in order to write useful programs, we almost always need the ability to check conditions and change the behavior of the program accordingly.
- **Conditional** statements give us this ability to affect the **flow of control**.
- The simplest form is the `if` statement. The Boolean expression after `if` is called the condition. If it is true, then the indented statement gets executed. If not, nothing happens.

```
if x > 0:
 print("x is positive")
```

### 1.4.2.2 Take a look at this example

```
animal = input("What is your favorite animal?")
if animal == 'cat' or 'dog':
 print("A great pet!")
else:
 print("Good choice")
```

### 1.4.2.3 Discussion

- Give students time to predict the output for various inputs in the above example.
- Discuss why the code is buggy
- fix it together as a class.

### 1.4.2.4 Demonstration

- Write out the syntax of the `if` statement on the board. Point out the Boolean expression(condition), the colon, and the indentation.
- Ask students if they recall what `else` went along with the `if` statement when they used it in Snap!
- `else` is used when there are two possibilities and the condition determines which one gets executed.
- Demonstrate the syntax of `else`
- Describe the `elif` statement
- Sometimes there are more than two possibilities and we need more than two branches. One way to express a computation like that is a chained conditional:
- Demonstrate the syntax of `elif`

## 1.4.3 3. Lab

- Students convert the triangle program written in Snap! into Python.
- Students must also write a program that simulates a list index using `if` statements.

## 1.4.4 4. Debrief

- Check student progress and completion of the lab, wrap up by taking any final questions.
- Have students write down a couple of learnings that they stood out to them today in their notebooks.

## **1.5 Accommodation/Differentiation**

---

Use the following as an extension activity for students that are moving quickly:

- Convert and finish the following SNAP Vending Machine program in Python.

*Vending Machine*

If students are moving quickly, this lesson can move onto lists.

## **1.6 Forums discussion**

---

[Lesson 2.03: Conditionals \(TEALS Discourse Account Required\)](#)

*formatted by [Markdeep 1.093](#)* 

# TEALS Program

[Home](#) | [Curriculum Map](#) | [Additional Readings](#) | [Change Log](#)

## 1 Do Now 2.03

---

In the console, create a schedule program. Given the hour of the day print out where you should be. If you're not doing anything else you should be "sleeping".

```
```python >>> What hour? 12pm >>> You should be at lunch! ````
```

1.1 In your Notebook

1.1.1 Answer the following

1. How did you accomplish this? Did you feel like something was missing in your program?
2. What if you wanted to add in a weekly functionality? For instance, maybe on Tuesday at 4pm you are at soccer practice, but on Thursday at 4pm you are at math club!
3. How would you implement this in your program?

formatted by [Markdeep 1.093](#) 

TEALS Program

[Home](#) | [Curriculum Map](#) | [Additional Readings](#) | [Change Log](#)

1 Lesson 2.04: Lists

1.1 Learning Objectives

Students will be able to...

- Define and identify: **list, item, index, integer**
- Be able to access items from a list using the index
- Create lists of different types
- Use the length function

1.2 Materials/Preparation

- [Do Now](#)
- [Lab - College Chooser \(printable lab document\)](#) ([editable lab document](#))
- Solution (access protected resources by clicking on "Additional Curriculum Materials" on the [TEALS Dashboard](#))
- [Associated Readings 2.4](#)
- Read through the do now, lesson, and lab so that you are familiar with the requirements and can assist students

1.3 Pacing Guide

Duration	Description
5 Minutes	Do Now
10 Minutes	Lesson
35 Minutes	Lab
5 Minutes	Debrief

1.4 Instructor's Notes

1.4.1 1. Do Now

- Students follow instructions to create lists and use the `len` function.

1.4.2 2. Lesson

1.4.2.1 Instruction

- Ask students to recall what a list is, and how lists were used in Snap!
- A **list** is a sequence of values. In a list, they can be any type. The values in a list are called elements or **items**.
- In Python, to create a list you must enclose items in square brackets.
- Emphasize that you can have lists of any type (int, float, string, etc). You can even have lists within lists (more on that later...)

1.4.2.2 Discussion

- Ask students what `len` does when they used it in the Do Now.
- Ask students how they tried to print the first item from a list. Was this what they were expecting?

1.4.2.3 More Instruction

- **index**: a map from the position in the list to the element stored there.
- **0-index**: lists are 0 indexed. So the first element in the list is at the 0-index
- **Out-of-bounds**: what happened when you tried to index into a list that was too long?

1.4.2.4 More Discussion

- Ask students how they would access the last item in a list of unknown length. (Use the `length` function!)
- Ask students to write on the board how they got the last element of a list. Ask another student to write how they would get the second to last element of the list and so on.

1.4.2.5 Demonstration

- After accessing any list element you can change it. Take a moment to demonstrate this syntax before starting the lab.

1.4.3 3. Lab

- Practice accessing and updating items in a list
- Implement program from last lab using lists
- Create a quiz program

1.4.4 4. Debrief

- Check student progress and completion of the lab, wrap up by taking any final questions.

1.5 Accommodation/Differentiation

If students are moving quickly, you can introduce the topic of nested lists. Start off with a simple nested list like `['a', 'b', 'c', [1, 2, 3]]`. Ask the students to guess the length. Ask the students to guess how they would

access the item '1' from that list!

1.6 Forum discussion

Lesson 2.04: Lists (TEALS Discourse Account Required)

formatted by Markdeep 1.093 

TEALS Program

[Home](#) | [Curriculum Map](#) | [Additional Readings](#) | [Change Log](#)

1 Do Now 2.04

1.1 1. In your console

1.1.1 Type in the following

```
a_list = ['a', 'b', 'c', 'd', 'e']
print(len(a_list))
```



1.2 In your Notebook

1.2.1 Answer the following

1. What type do you think a_list is? (Hint: look at the name of the variable)
2. What does len do?
3. Brainstorm how you would print the first element from a_list?

1.3 2. In your Console

1.3.1 Type the following

```
a_list = ['a', 'b', 'c', 'd', 'e']
print(a_list[0])
print(a_list[1])
print(a_list[2])
print(a_list[3])
print(a_list[4])
print(a_list[5])
print(a_list[6])
```



1.4 Continue In your Notebook

Explain what happens in the program

TEALS Program

[Home](#) | [Curriculum Map](#) | [Additional Readings](#) | [Change Log](#)

1 Lab 2.04 - Food Chooser

1.1 1. In your notebook

For each example below, predict what will be printed. Next, run the program and confirm what was output.

1.1.1 Example 1

```
a = ['a', 'b', 'c', 'd', 'e']
print(a[0])
print(a[3])
```



1.1.2 Example 2

```
a = ['a', 'b', 'c', 'd', 'e']
print(a[len(a) - 3])
```



1.1.3 Example 3

```
a = ['a', 'b', 'c', 'd', 'e']
print(a[len(a) - 6])
```



1.1.4 Example 4

```
a = ['a', 'b', 'c', 'd', 'e']
a[3] = 'haha'
print(a)
```



1.2 2. Create this game again using lists and indexes. Updated rules below

- Declare 10 prizes (prize0, prize1, prize2 at the top of your file), but store them all in a list.
- User picks a number.
- Print prize associated with the door user picked.

1.3 3. Create a quiz

Create a food quiz using lists and indexes.

1. List of 6 different foods
2. Ask the user 8 vague questions to find out what their favorite food it out of the list

3. Update the score and print their top 2 favorite foods

Hint: google how to find the biggest number in a list python

[Starter code here](#)

1.4 Bonus

Research nested lists and work through the following:

1.4.1 Bonus Example 1

```
a = ['a', 'b', 'c', ['d', 'e']]  
print(len(a))
```



1.4.2 Bonus Example 2

```
a = ['a', 'b', 'c', ['d', 'e']]  
b = a[3]  
print(b)
```



1.4.3 Bonus - In your Notebook

How would you access 'd' from the list a?

formatted by [Markdeep 1.093](#) 

TEALS Program

[Home](#) | [Curriculum Map](#) | [Additional Readings](#) | [Change Log](#)

1 Lesson 2.05: Lists Part 2

1.1 Learning Objectives

Students will be able to... * Define and identify: **index, slice, append, pop, remove** * Slice a list * Add and remove elements from a list

1.2 Materials/Preparation

- [Do Now](#)
- [Lab - Tic-Tac-Toe \(printable lab document\)](#) ([editable lab document](#))
- [Associated Readings 2.5](#)
- Read through the do now, lesson, and lab so that you are familiar with the requirements and can assist students

1.3 Pacing Guide

Duration	Description
10 Minutes	Do Now
10 Minutes	Lesson
30 Minutes	Lab
5 Minutes	Debrief

1.4 Instructor's Notes

1.4.1 1. Do Now

- Students may need more time than usual to fully go through this lesson's Do Now.

1.4.2 2. Lesson

1.4.2.1 Instruction

- Ask students what doing `a[0:2]` in the Do Now did.
- Define **slicing**: a list operation that gives back a list starting from the index to the left of the colon and going up to the index to the right of the colon.
- Note that slicing doesn't exist in Snap!
- Ask students what the list would return if you did `a[1:2]`.
- Explore the differences between `remove` and `pop`, asking for student input.
- Ask students what the plus sign and `append` do to a list.
- Ask students to write down or brainstorm how they would represent a Tic-Tac-Toe board using lists.
- Go over the `list in` operation. Ask what the return value is (Boolean expression).
- Have students practice using the `in` operation in an if statement.

1.4.2.2 Demonstration

- Create a Tic-Tac-Toe board with students in class.

1.4.3 3. Lab

- Students practice slicing, adding, and removing elements from some given lists.
- Students create a single move Tic-Tac-Toe game

1.4.4 4. Debrief

- Check student progress and completion of the lab, wrap up by taking any final questions.

1.5 Accommodation/Differentiation

- If students are moving quickly, start the next topic of using a while loop as a game loop. Explore the concept of keeping score for the game.
- If students are moving slowly then spend an extra day reviewing lists and completing lab activities.

1.6 Quiz Option

There is also an opportunity for a quiz after the game loop lesson and before the project.

1.7 Forum discussion

[Lesson 2.05: Lists 2 \(TEALS Discourse Account Required\)](#)

TEALS Program

[Home](#) | [Curriculum Map](#) | [Additional Readings](#) | [Change Log](#)

1 Do Now 2.05

1.1 Part 1 - Open up the console

1.1.1 Type the following example code using interactive mode

1.1.2 Example 1

```
>>> a = [123, 'First Item', 456, 'Second Item']
>>> b = a[0:2]
>>> print(a)
>>> print(b)
```

//

1.1.3 In your notebook

1.1.3.1 Respond to the following

1. What happens to a?
2. What is b?
3. What type is b?

1.1.4 Example 2

```
>>> a = [123, 'First Item', 456]
>>> a.remove('First Item')
>>> print(a)
```

//

1.1.5 Continue in your notebook

1.1.5.1 Respond to the following questions

1. What does remove do?
2. What would the length of this list be after remove?

1.1.6 Example 3

```
>>> a = [123, 'First Item', 456]  
>>> a.pop()  
>>> print(a)
```



1.1.7 Continue your responses in your notebook

1. what does pop do?
2. What is the difference between remove and pop?

1.1.8 Example 4

```
>>> a = [123, 'First Item', 456, 'Second Item']  
>>> b = a + ['Third Item']  
>>> print(a)  
>>> print(b)
```



1.1.8.1 Continue your responses to the following in your notebook

1. What happens to a?
2. What is b?
3. What type is b?

1.1.9 Example 5

```
>>> a = []  
>>> print(len(a))  
>>> a.append('First Item')  
>>> print(a)
```



1.1.9.1 In your Notebook

1. What was the length of []?
2. What does append do? What would the length be after append?

TEALS Program

[Home](#) | [Curriculum Map](#) | [Additional Readings](#) | [Change Log](#)

1 Lab 2.05 - Tic-Tac-Toe

1.1 In your Notebook

1.1.1 1. Predict what will be printed. Then run the program and confirm

1.1.1.1 Example 1

```
a = ['a', 'b', 'c', 'd', 'e']
print(a[0:3])
print(a[1:4])
```

//

1.1.1.2 Example 2

```
a = ['a', 'b', 'c', 'd', 'e']
print(a[1:len(a) - 3])
```

//

1.1.1.3 Example 3

```
a = ['a', 'b', 'c', 'd', 'e']
b = a.remove('b')
print(a)
print(b)
```

//

1.1.1.4 Example 4

```
a = ['a', 'b', 'c', 'd', 'e']
a[0] = 'haha'
b = a.pop()
print(a)
print(b)
```

//

1.1.1.5 Example 5

```
a = ['a', 'b', 'c', 'd', 'e']
b = a + ['abc']
print(a)
print(b)
```

//

1.1.1.6 Example 6

```
a = ['a', 'b', 'c', 'd', 'e']
b = a.append('f')
print(a)
print(b)
```



1.1.2 2. In your Console

We are going to start implementing Tic-Tac-Toe using a single list.

1. The user picks a location on the board according to the number:

tic-tac-toe

2. Depending on the position that the user inputs, update the position of the board to an "X" to reflect that.
3. Print the updated board out, but don't worry about making it look pretty.
4. Only need to implement one turn of the game

formatted by [Markdeep 1.093](#) 

TEALS Program

[Home](#) | [Curriculum Map](#) | [Additional Readings](#) | [Change Log](#)

1 Lesson 2.06: Game Loop

1.1 Learning Objectives

Students will be able to...

- Define and identify: **while loop**
- Use a while loop to simulate game play

1.2 Materials/Preparation

- [Do Now](#)
- [Lab - Tic-Tac-Toe Revisited \(printable lab document\)](#) ([editable lab document](#))
- [Associated Readings 2.7](#)
- Read through the do now, lesson, and lab so that you are familiar with the requirements and can assist students

1.3 Pacing Guide

Duration	Description
5 Minutes	Do Now
10 Minutes	Lesson
35 Minutes	Lab
5 Minutes	Debrief

1.4 Instructor's Notes

1.4.1 1. Do Now

- Students experiment with creating while loops.

1.4.2 2. Lesson

1.4.2.1 Instruction

- Ask students what the while loops that they created did.
- Go over the syntax of while loops. Use this as an opportunity to remind students of **Boolean expressions**.

1.4.2.2 Discussion

- Ask: how might while loops be useful?
- Ask students to consider how they could write a loop using user input, using the following scenario:
- What if you wanted the loop to stop when the user inputs "quit"?

1.4.2.3 Activity

- Have the students think about and write a solution

1.4.2.4 Student Share

- call students up to the board to write out how they solved it.

1.4.3 3. Lab

- Students work to create a Tic-Tac-Toe game that allows turns (up to 9), building on their previous work.

1.4.4 4. Debrief

- Check for completion and understanding of the lab.

1.5 Accommodation/Differentiation

If there is extra time, have students start reading through the project specs and thinking about how they will apply what they have learned this unit to complete the project.

1.6 Forum discussion

Lesson 2:06: Game Loop (TEALS Discourse Account Required)

TEALS Program

[Home](#) | [Curriculum Map](#) | [Additional Readings](#) | [Change Log](#)

1 Do Now 2.06

1.1 1. In your Notebook

1.1.1 Answer the following

1. How would you print out something 10 times? What about 100? What about 1,000?
2. Can you remember something from Snap! that might allow that?

1.2 2. Open up the console

1.2.1 Type the following code using interactive mode

```
```python while True: print('a')
```

## 3. In your Notebook

### Write responses for the following

1. What happens `when` you run this code?
2. Try using other Boolean expressions instead of `True` (e.g. `4 > 5` and `9 != 2`), and explore what happens.
3. How would you `print` out something `10 times`? What about `100`? What about `1,000`?
4. Can you remember something from Snap! that might allow that?

## 4. Open up the console

```
Type and run the following code (Remember this from Unit 2?)
if animal in ['cat', 'dog']:
 print("A great pet!")
```





# TEALS Program

[Home](#) | [Curriculum Map](#) | [Additional Readings](#) | [Change Log](#)

## 1 Lesson 2.07: Project

---

### 1.1 Learning Objectives

---

Students will be able to... \* Use knowledge of lists, Booleans, conditionals, and while loops to create a text-based adventure game.

### 1.2 Materials

---

- Project Spec - Text Monster ([printable project Spec](#)) ([editable project spec](#))
- Project Spec - Canadian Variation
- Project Spec - Todo List ([printable Alternate project Spec](#)) ([editable Alternate project spec](#))
- Text Monster Starter Code
- [Editable Grading Rubric](#)
- Sample Solutions (access protected resources by clicking on “Additional Curriculum Materials” on the [TEALS Dashboard](#))

### 1.3 Preparation

---

- Read through the [Associated Readings 2.7](#)
- Try creating your own variation on the Text Monster code so you are familiar with the potential challenges and bugs your students will hit.
- Review [4 Steps to Solve Any CS Problem]
- Update the Project Spec of your selected project as needed to meet your grading requirements

#### 1.3.1 Day 1 Pacing Guide

Duration	Description
10 Minutes	Project Overview/Demo
40 Minutes	Design
5 Minutes	Debrief

#### 1.3.2 Days 2 – 9 Pacing Guide

Duration	Description
10 Minutes	Review
40 Minutes	Project Work
5 Minutes	Debrief

## 1.4 Instructor's Notes

---

### 1.4.1 1. 4 Steps to Solve Any CS Problem

- Review [4 Steps to Solve Any CS Problem]

### 1.4.2 2. Project Overview/Demo

- Distribute the project spec to all students and walk them through the goals and requirements of the project.
- Show a demo of a completed project.
- Go over specific design considerations from the project spec:
- Introduce the concept of global variables and how they will be useful here.
- Identify the importance of the “User Pocket” and how to use a list along with append and remove for this information.

### 1.4.3 3. Design

- Have students stay at their desks and write down what lists they'll need.
- They should break up the project into parts: parsing user input, keeping track of players position, returning what is at the player's position .

### 1.4.4 4. Debrief/Review

- During discussion and wrap up at the end of class, get a feeling for where students are in the project.
- During the review the next morning cover the topics/areas that students are struggling on and present tips, suggestions, and goals for that day.

## 1.5 Accommodation/Differentiation

---

- Make sure to do status checks with all students throughout the project.
- Identify students that are struggling on the project after the first few days and provide additional scaffolding & support as needed.
- For any students that are advancing rapidly through the project, give them extension ideas such as adding a new feature or floor to the game.
- Advanced students can also be paired as tutors/helpers with struggling students.

## 1.6 Grading

---

### **1.6.1 Objective Scoring Breakdown**

#### **Editable Grading Rubric**

Student correctly identifies data types (Lesson 2.01) - Assessed in Unit 3

<b>Points</b>	<b>Percentage</b>	<b>Objective</b>	<b>Lesson</b>
3	12%	Student correctly uses conditionals to maintain flow of control	2.02, 2.03
9	36%	Student correctly uses lists	2.04 2.05
3	12%	Student can correctly use the while loop	2.06
5	20%	Student can decompose a problem to create a program from a brief	
5	20%	Student uses naming/ syntax conventions and comments to increase readability	
<b>25</b>	<b>Total Points</b>		

### **1.6.2 Scoring Consideration**

You may need to adjust the points in order to fit your class. Treat the percentages as a guide to determine how to weight the objectives being assessed.

## **1.7 Forum discussion**

Lesson 2.07: Text Game (TEALS Discourse Account Required)

formatted by Markdeep\_1.093 

# TEALS Program

[Home](#) | [Curriculum Map](#) | [Additional Readings](#) | [Change Log](#)

## 1 Project 2: Text Monster Game

---

Using Python, students will use casting, Boolean expressions, lists and while loops to create a text-based adventure game!

### 1.1 Overview

---

This game takes place in a three story dungeon. The user has to traverse the levels in search of the prize. Along the way they collect items and fight monsters. On each move the user has seven possible commands: left, right, up, down, grab, fight, help. If the input is invalid (not one of these commands,) the game will let the user know. Otherwise, the game will execute the user's command. The goal of the game is to collect the prize guarded by the boss monster.

### 1.2 Details

---

#### 1.2.1 Behavior

```
What would you like to do? left
You see sword
What would you like to do? grab
Picked up sword
What would you like to do? left
You see nothing
What would you like to do? left
You see monster
What would you like to do? fight
You defeated the monster!
What would you like to do? left
You see stairs down
What would you like to do? down
You see nothing
What would you like to do? right
You see stairs up
What would you like to do? left
You see nothing
What would you like to do? left
Sorry cant go that way.
What would you like to do? up
Cannot go that way.
What would you like to do? right
You see nothing
What would you like to do? up
Cannot go that way.
What would you like to do? right
You see stairs up
What would you like to do? up
You see nothing
What would you like to do? right
You see nothing
What would you like to do? right
You see nothing
```

What would you like to do? right  
You see nothing  
What would you like to do? up  
Cannot go that way.  
What would you like to do? left  
You see nothing  
What would you like to do?

//

#### **1.2.1.1 The game has three floors**

- Each floor is made up of five rooms, arranged in a line from left to right.
- A room can contain: a sword, a monster, magic stones, up-stairs, down-stairs or nothing.

#### **1.2.1.2 At the start of the game, the user is placed in one of the rooms**

#### **1.2.1.3 Movement**

- The user can try to move to the left room or right room.
- If there is no room in that direction, the game should report this.
- The user can also move upstairs or downstairs if the room contains an up-staircase or a down-staircase.

#### **1.2.1.4 Contents of rooms**

- The game prints out the contents of the current room after every command.
- The user can grab swords or magic stones if they walk into a room with them.
- The sword or stones are no longer in the room once grabbed.

#### **1.2.1.5 Monsters guard some rooms**

- The user can use a sword to defeat a monster using the fight command.
- The sword and monster disappear after fighting.
- If they have no sword, the user can exit in the direction from which they came.
- If the user fights without a sword, they will be defeated and the game will end.
- If they try to walk past a monster, they will be killed and the game will end.
- A sword and magic stones are required to defeat the boss monster.

### **1.2.2 Implementation Details**

- The game should be implemented using lists.

#### **1.2.2.1 User Items**

- Use a list to keep track of the user's items.
- At the beginning of the game it should be empty.
- A maximum of three items can be held at once.

#### **1.2.2.2 Monsters**

- There should be three regular monsters placed throughout the game.

- These require a sword to defeat.
- There should be a boss monster in the room just before the room that contains the prize.
- This monster requires magic stones and a sword to defeat.

### 1.2.2.3 Movement Implementation

- The user can only go up if there is an up-staircase
- The user can only go down if there is down-staircase.
- The program should not allow the user to run past a monster,
- The program should not allow the user to go up, down or past bounds of the game.

### 1.2.2.4 Win/Lose

- The game is won when the player grabs the prize.
- The game is lost if the user
- fights a monster without a sword
- fights the boss monster without a sword and stones
- tries to move past a monster

## 1.2.3 Design Considerations

### 1.2.3.1 Game Board

The game board is the basis of the game. The following is a way to think of a smaller game board as a set of three lists: one for each floor.

```
floor_1 = ['nothing', 'nothing', 'stairs up']
floor_2 = ['nothing', 'nothing', 'stairs up']
floor_3 = ['prize', 'nothing', 'nothing'] //
```

The above code has each floor being its own lists. Feel free to use a different implementation, but this should work for our purposes.

### 1.2.3.2 User Position

It will be useful to keep track of the user's position through a variable.

```
user_room = 0
user_floor = floor_1 //
```

This would put the user at the position of the first room of the first floor.

### 1.2.3.3 Validating User Input

You will need to check the input of the user to make sure it is valid for the current game state:

```
if user_input == "down":
 current_room = user_floor[user_room]
 if current_room != "stairs down":
 print("Can't go downstairs; there are no stairs.") //
```

## 1.3 Extra Credit

- Add the command `run`, which allows a player to run past a monster instead of fighting. This should work 40% of the time. (Hint: Research the random library.)
- Implement the board using nested lists (each item of the list is a list.)

*formatted by [Markdeep 1.093](#)* 

# TEALS Program

[Home](#) | [Curriculum Map](#) | [Additional Readings](#) | [Change Log](#)

## 1 Alternate Project 2: TODO List

---

Created By Brian Weinfeld

Using Python, you will create an interactive TODO list that allows users to add and remove tasks from a list.

### 1.1 Overview

---

One of the most common tasks we use computers for is to store and track data. Perhaps the most simple form of this task is the TODO list. A TODO list is simply a list of information (for example, chores that needs to be done) that the user wants to track. They can add tasks to the list, remove them when they are complete and examine the list as they progress. Have you ever used a notes app on your phone?

### 1.2 Details

---

#### 1.2.1 Behavior

```
Welcome to your TODO list
What would you like to do? (add, remove, list, exit) list
[]
```

```
What would you like to do? (add, remove, list, exit) add
What would you like to add to your list? go to gym
Successfully added
What would you like to do? (add, remove, list, exit) list
['go to gym']
```

```
What would you like to do? (add, remove, list, exit) add
What would you like to add to your list? buy food
Successfully added
What would you like to do? (add, remove, list, exit) list
['go to gym', 'buy food']
```

```
What would you like to do? (add, remove, list, exit) add
What would you like to add to your list? go to gym
This is already on your list!
What would you like to do? (add, remove, list, exit) list
['go to gym', 'buy food']
```

```
What would you like to do? (add, remove, list, exit) remove
What would you like to remove? go to gym
Successfully removed
What would you like to do? (add, remove, list, exit) list
['buy food']
```

```
What would you like to do? (add, remove, list, exit) remove
What would you like to remove? clean bedroom
This is not in the list!
What would you like to do? (add, remove, list, exit) list
['buy food']
```

What would you like to do? (add, remove, list, exit) exit  
Goodbye



### 1.2.2 Implementation Details

- The program offers the user 4 options. **add** will add elements to the TODO list. **remove** will remove elements from the list. **list** will display the entire list and **exit** will exit the program.
- After a user selects **add** they should then be prompted for the item they want added to the list. Be sure to check before you add the item to the list. You don't want to add the item twice!
- After a user selects **remove** they should then be prompted for the item they want removed from the list. Be sure to check before removing the item from the list. You should let the user know if they are trying to remove an item that isn't in the list!

### 1.2.3 Challenge

This section contains additional components you can add to the project. These should only be attempted **after** the project has been completed.

- It is a bit inconvenient to have to type two commands to add or remove an element from the list. Modify the program so that tasks can be added or removed from the TODO list with a single command like **add go to gym** or **remove go to gym**.

```
Welcome to your TODO list
What would you like to do? (add, remove, undo, list, exit) add go to gym
Successfully added
What would you like to do? (add, remove, undo, list, exit) list
['go to gym']
```

```
What would you like to do? (add, remove, undo, list, exit) add clean bedroom
Successfully added
What would you like to do? (add, remove, undo, list, exit) list
['go to gym', 'clean bedroom']
```

```
What would you like to do? (add, remove, undo, list, exit) remove bedroom
This is not in the list!
What would you like to do? (add, remove, undo, list, exit) remove clean bedroom
Successfully removed
What would you like to do? (add, remove, undo, list, exit) list
['go to gym']
```

- It is helpful to have a feature to immediately remove the most recently added element to the list. This is often because the user made a mistake in adding the element to the list in the first place. Add an **undo** option to the program that will remove the most recently added item. If the item is no longer in the list, print an error.

```
Welcome to your TODO list
What would you like to do? (add, remove, undo, list, exit) add go to gym
Successfully added
What would you like to do? (add, remove, undo, list, exit) add clean bedroom
Successfully added
What would you like to do? (add, remove, undo, list, exit) list
['go to gym', 'clean bedroom']
```

```
What would you like to do? (add, remove, undo, list, exit) undo
Successfully undid last add.
What would you like to do? (add, remove, undo, list, exit) list
['go to gym']
```

```
What would you like to do? (add, remove, undo, list, exit) undo
Cannot undo. You already removed this item
What would you like to do? (add, remove, undo, list, exit) list
['go to gym']
```



#### 1.2.4 Super Challenge

The super challenge will require knowledge that has not been taught yet. You will need to do additional research on your own. Good luck!

It is possible to track enough information so that the **undo** command can be called repeatedly. Modify the **undo** command so that it will always work by removing the most recently added element in the TODO list that is still in the list. The only time the command should do nothing is when the TODO list is empty.

*formatted by [Markdeep 1.093](#)* 

# TEALS Program

[Home](#) | [Curriculum Map](#) | [Additional Readings](#) | [Change Log](#)

## 1 Unit 3 - Functions

---

### 1.1 Essential Questions

---

- What are some built in functions in Python?
- How do you apply printing and returning values in a function?
- How do you define your own functions in Python?
- What is the difference between global and local variables?"

### 1.2 Pacing Guide

---

#### 1.2.1 Timing

1 Day = 50 minute class period

Lesson	Days	Resources
3.01: Built In Functions	1	<a href="#">3.01 Slide Deck</a>
3.02: User-Defined Functions	2	<a href="#">3.02 Slide Deck</a>
3.03: Return vs Print	1	<a href="#">3.03 Slide Deck</a>
3.04: Debugging and Scope	1	<a href="#">3.04 Slide Deck</a>
3.05: Oregon Trail	9	<a href="#">3.05 Slide Deck</a>
<b>Total Days</b>	14	
<b>Total Minutes</b>	700	

### 1.3 Key Terms

---

- Function
- Arguments
- Calling Functions
- Importing
- Returning
- Abstraction

- Def
- Return
- None
- Void
- Scope
- aliasing
- stack trace

*formatted by [Markdeep 1.093](#) 🎨*

# TEALS Program

[Home](#) | [Curriculum Map](#) | [Additional Readings](#) | [Change Log](#)

## 1 Lesson 3.01: Built-In Functions

---

### 1.1 Learning Objectives

---

Students will be able to...

- Define and identify: **function, arguments, calling, importing, returning**
- Call the built-in randint function, using arguments
- Utilize code other people have written in the Python documentation
- Understand the difference between printing and returning

### 1.2 Materials/Preparation

---

- [Do Now](#)
- [Lab - Magic 8-Ball \(printable lab document\)](#) ([editable lab document](#))
- [Associated Reading](#)
- Read through the do now, lesson, and lab so that you are familiar with the requirements and can assist students.

### 1.3 Pacing Guide

---

Duration	Description
5 Minutes	Do Now
10 Minutes	Lesson
35 Minutes	Lab
5 Minutes	Debrief

### 1.4 Instructor's Notes

---

#### 1.4.1 1. Do Now

- Give students 3-4 minutes to follow the instructions on the Do Now page.
- Debrief the answers to the questions on the Do Now by calling on students to respond.

## 1.4.2 2. Lesson

### 1.4.2.1 Build Your Own Blocks (Snap) vs Functions (Python)

- Ask students to recall how they built custom blocks in Snap!

*Snap Custom Block*

- Function:** a named sequence of statements. You can use functions to perform complex calculations, graphical operations, and various other purposes. When you define a function, you specify the name and the sequence of statements. Later, you can "call" the function by name.
- In SNAP! functions are blocks

### 1.4.2.2 Function Contract

A function definition is like a contract: you tell the programmer what elements the function expects (name and type of arguments) and the function will perform its purpose. It is good practice to use a comment to specify the purpose and contract of a function, including the type of value it returns, if it returns a value.

\* Name:  
\* Purpose:  
\* Arguments:  
\* Returns:



- Explain that we have already gotten used to **calling** functions like `type()` and `print()`.
- Ask students how they would create a random number generator.
- Sounds hard! Luckily someone has already done that: the random library (essentially a bunch of code written by someone else) which has many associated functions.

### 1.4.2.3 Back to the Do Now

- Remind students what they saw in the Do Now - how to get a random integer: `randint(0, 10)`.
- Identify the 0 and 10 in this example as **arguments**, or values passed into the function.
- Ask students what the argument is when we use `print` or `type`
- `randint` gives back a value that you might want to store - this is called **returning**. If nothing is given back, the return value is `None`.

### 1.4.2.4 More on Function Contracts

- Functions have a contract: you write down the name, purpose, arguments with their type, and the return type expected.
- Ask students what the contract of `randint` is.

\* Name: `randint`  
\* Purpose: generate a pseudo-random integer N such that  $a \leq N \leq b$   
\* Arguments: 2 values of type integer: `a` and `b`  
\* Returns: integer



- Since `randint` is written by someone else there is a place where that contract is written out - **Documentation**. Have students begin the lab, which will instruct them to find the Python documentation for the random library.

## 1.4.3 3. Lab

- Students look through `random` library documentation, practice importing different random functions and using them.
- Create a Magic 8-ball program using a list and `randint`.

#### 1.4.4 4. Debrief/Exit Ticket

- In their notebooks, have students right down 2 things they learned today to reinforce learning.

#### 1.5 Accommodation/Differentiation

---

If students are moving quickly, find another library to import from (see **bonus** in the lab) OR allow students to move on to creating their own functions.

#### 1.6 Forum discussion

---

Lesson 3.01: Built-In Functions (TEALS Discourse Account Required)

formatted by Markdeep 1.093 

# TEALS Program

[Home](#) | [Curriculum Map](#) | [Additional Readings](#) | [Change Log](#)

## 1 Do Now 3.01

---

### 1.1 In your Console

---

#### 1.1.1 Type the following code

```
import random
random.randint(0, 3)
random.randint(0, 3)
print(random.randint(0, 3))
print(random.randint(0, 3))
print(random.randint(0, 3))
```



### 1.2 In your Notebook

---

#### 1.2.1 Respond to the following

1. What does randint do?
2. What do the values 0 and 3 do? Try changing those numbers, rerun the program, and write down what changed.
3. What is the difference between `random.randint(0,3)` and `print(random.randint(0,3))`?

# TEALS Program

[Home](#) | [Curriculum Map](#) | [Additional Readings](#) | [Change Log](#)

## 1 Lab 3.01 - Magic 8-Ball

---

Practice importing random\*\* — Use randint with different arguments. Simulate a dice roll, printing out to the user what number they rolled.

Look at the [documentation] of the random library — Experiment with another function (not randint) that returns a value.

### 1.1 Create a program that simulates a [magic 8-ball]

---

1. Store all of the 8-ball's possible responses (shown below) in a list
2. Have the program prompt the user to ask the magic 8-ball a question
3. then return and print a random response.

#### 1.1.1 Magic 8-Ball Response Examples

- Outlook is good
- Ask again later
- Yes
- No
- Most likely no
- Most likely yes
- Maybe
- Outlook is not good

### 1.2 Video Explanation

---

[

](<https://www.youtube.com/watch?v=vZRrg6Nl-1E>)

*Magic 8 Ball*

### 1.3 Bonus

---

Research the math library and create a program that finds the length of the hypotenuse of a right triangle given two sides.

# TEALS Program

[Home](#) | [Curriculum Map](#) | [Additional Readings](#) | [Change Log](#)

## 1 Lesson 3.02: User-Defined Functions

---

### 1.1 Learning Objectives

---

Students will be able to...

- Define and identify: **abstraction, def**
- Create functions

### 1.2 Materials/Preparation

---

- [Do Now](#)
- [Lab - Birthday Song & Random Cards \(printable lab document\)](#) ([editable lab document](#))
- [Associated Reading](#)
- Read through the do now, lesson, and lab so that you are familiar with the requirements and can assist students.

### 1.3 Pacing Guide

---

Duration	Description
5 Minutes	Do Now
10 Minutes	Lesson
35 Minutes	Lab
5 Minutes	Debrief

### 1.4 Instructor's Notes

---

#### 1.4.1 1. Do Now

- Students should take 5 minutes to follow the instructions on the Do Now in order to create/manipulate a user-defined function.

## **1.4.2 2. Lesson**

### **1.4.2.1 Abstraction**

- Abstraction is managing the complexity of a program by removing details and pushing them down to a lower level
- Ask students to brainstorm why a function might be useful in programming.
  - Less repeated code.
  - Breaking the problem up into smaller pieces and solving each piece

### **1.4.2.2 Demonstration**

- Demonstrate to students how you create a function using `def`, calling out the syntax and where arguments would go.
- Ask students how they would call your example function.
- Have students practice making a function that takes two arguments, adds them together, and returns the sum.

### **1.4.2.3 Function Contracts**

- Introduce the concept of a function contract using `#`, which adds a comment (non-executed line of code)
- The function contract should
  - specify the name
  - explain the purpose
  - list what arguments it takes in and the types of those arguments
  - specify the return type

## **1.4.3 3. Lab**

- Practice making a function that will take in a name as an argument and output the 'happy birthday song' to that name.
- Create a function that randomly selects 5 cards from a deck of cards (repeating allowed).

## **1.4.4 4. Debrief**

- Check student progress and completion of the lab, wrap up by taking any final questions.

## **1.5 Accommodation/Differentiation**

---

If students are moving quickly, they could go back and use functions to improve an old project.

## **1.6 Forum discussion**

---

[Lesson 3.02: User-Defined Functions \(TEALS Discourse Account Required\)](#)



# TEALS Program

[Home](#) | [Curriculum Map](#) | [Additional Readings](#) | [Change Log](#)

## 1 Do Now 3.02

---

### 1.1 In your Console

---

#### 1.1.1 Type the following code

```
def my_function():
 print("THIS IS MY FUNCTION!")
```



### 1.2 In Your Notebook

---

#### 1.2.1 Respond to the following

1. What does `my_function` do?
2. How would you call the function? Practice calling `my_function` and checking that it does what you expect it to.
3. How would you add arguments to `my_function`?

formatted by [Markdeep 1.093](#)

# TEALS Program

[Home](#) | [Curriculum Map](#) | [Additional Readings](#) | [Change Log](#)

## 1 Lab 3.02 - Birthday Song & Random Cards

---

### 1.1 Lab Exercise 1

---

Create a function, `birthday_song`, that prints out the happy birthday song to whatever name is input as an argument. The contract should be:

```
Name: birthday_song
Purpose: birthday_song prints out a personalized birthday song
Arguments: name, string
Returns: none
def birthday_song(name):
 #your code goes here
```



### 1.2 Lab Exercise 2

---

- Create a function that randomly picks 5 cards from a deck
- The cards can repeat

Write out the contract for this function using the list.

```
number = ['A', '2', '3', '4', '5', '6', '7', '8', '9', '10', 'J', 'Q', 'K']
```



#### 1.2.1 Bonus

- Practice passing in lists as an argument to a function.
- What is different about passing in a list as an argument?
- Read about list aliasing in section 3.4 of the associated reading, and write down what is happening in this case.

# TEALS Program

[Home](#) | [Curriculum Map](#) | [Additional Readings](#) | [Change Log](#)

## 1 Lesson 3.03: Return vs Print

---

### 1.1 Learning Objectives

---

Students will be able to...

- Define and identify: **return, none**
- Explain and demonstrate the difference between printing and returning

### 1.2 Materials/Preparation

---

- [Do Now](#)
- [Lab - War \(Card Game\)](#) (printable lab document) (editable lab document)
- [Associated Reading](#)
- Read through the do now, lesson, and lab so that you are familiar with the requirements and can assist students.
- Note that this lesson may take two days.

### 1.3 Day 1 Pacing

---

Duration	Description
5 Minutes	Do Now
10 Minutes	Lesson
35 Minutes	Lab
5 Minutes	Debrief

### 1.4 Day 2 Pacing

---

Duration	Description
5 Minutes	Do Now
40 Minutes	Lab
5 Minutes	Debrief

## 1.5 Instructor's Notes

---

### 1.5.1 1. Do Now

- Students experiment with a function that returns a value, but they must add a print command to output that value.

### 1.5.2 2. Lesson

- Ask students about what they think the difference between returning and printing is.

#### 1.5.2.1 Student Sharing

- Get a volunteer to describe how they rewrote the code in the Do Now to get a value output.
- Ask a student to write the code on the board.

#### 1.5.2.2 Discussion

- Discuss the concept of the function contract again, explaining that the functions we will work with have both inputs and outputs.
- Returning is a concept in Snap!, just with a different name: reporting.

*Max Function including the reporter Block*

#### 1.5.2.3 Building a Structure Activity

- One student volunteer represents the `give_card` function.
- This student holds the deck of cards and stands by the board.
- On the board display the `give_card` function in code that only **prints** the value of a randomly chosen card.
- Students 'call' the student and request cards, which then the student follows the instructions and draws ('prints') the card on the board.
- Display a new `give_card` function that **returns** a card instead.
- Have students 'call' the function, however this time have the `give_card` student pass out the card when a student calls him/her.
- \* Debrief the activity and talk about what was learned.

### 1.5.3 3. Lab

- Given a shuffled deck list, students will create a program that plays the game 'War' with the user.

#### 1.5.4 4. Debrief

- Check student progress and completion of the lab, wrap up by taking any final questions.

### 1.6 Accommodation/Differentiation

---

As an extension activity, ask students to research the shuffle function and the functions associated with it.

### 1.7 Forum discussion

---

[Lesson 3.03: Return vs Print \(TEALS Discourse Account Required\)](#)

formatted by [Markdeep 1.093](#) 

# TEALS Program

[Home](#) | [Curriculum Map](#) | [Additional Readings](#) | [Change Log](#)

## 1 Do Now 3.03

---

### 1.1 In your Console

---

#### 1.1.1 Type the following code

```
import random
inputs: x (int), y (int)
outputs: int
50% returns sum of x and y, 50% returns product of x and y

def mystery_function(x, y):
 random_number = random.randint(0,1)
 if random_number > 0:
 z = x + y
 else:
 z = x * y
 return z
mystery_function(1, 2)
```



### 1.2 In your Notebook

---

#### 1.2.1 Answer the following questions

1. What happens when you run this code? How do you know what the result was?
2. Keeping the function the same, rewrite the code to print out the value that the function returns.

# TEALS Program

[Home](#) | [Curriculum Map](#) | [Additional Readings](#) | [Change Log](#)

## 1 Lab 3.03 - War (Card Game)

---

Create a program that lets a user play a **simplified** version of the card game '[War](#)'. In this version, the users will share a single deck of cards and cards will not be added back to the deck after they have been played.

### 1.1 Video Explanation of the Card Game War

---

[

](<https://youtu.be/G4DzhzDlXFM>)

*Card Game: War*

#### 1.1.1 Your game should

- Start with a given shuffled deck variable (shuffle function comes from python's random library, more details below)
- Ask for player1 and player2's names.
- Have a function `player_turn`, with the contract shown below:

```
name: player_turn
purpose: takes in a player name and draws/removes a card from the deck, prints "user drew card x", and returns the
value
Arguments: player_name as string, deck as list
returns: integer
```

//

- Have a function `compare_scores` that takes in the two integers representing the cards drawn and compares the card values. Make sure to write the contract for `compare_scores`!
- For simplicity Jacks will be represented as 11, Queens will be represented as 12, Kings will be represented as 13, and Aces will be represented as 14
- For simplicity the suit does not matter
- Include a while loop that keeps the game running until there are no cards in the deck.
- If there is a tie, there is "war". Take the next two cards an whoever wins that gets all four cards (including the previous tied cards). If there is another tie, continue taking the next two cards until there a winner. The winner takes all the "war" cards.
- Keep track of the score.
- Player who won the most number of cards wins.
- Declare the name of the winner and final score at the end of the game.

### 1.2 Sample Output

---

Player 1's name: Pat Player 2's name: Sam

Pat drew card 8 Sam drew card 9 Sam has high card Pat: 0 Sam: 2

Pat drew card 9 Sam drew card 8 Pat has high card Pat: 2 Sam: 2

Pat drew card 7 Sam drew card 7 War Pat: 2 Sam: 2

Pat drew card 5 Sam drew card 6 Sam has high card Sam wins war of 4 cards Pat: 2 Sam: 6

Pat drew card 13 Sam drew card 14 Sam has high card Pat: 2 Sam: 8  
Pat drew card 6 Sam drew card 12 Sam has high card Pat: 2 Sam: 10  
Pat drew card 4 Sam drew card 8 Sam has high card Pat: 2 Sam: 12  
Pat drew card 12 Sam drew card 2 Pat has high card Pat: 4 Sam: 12  
Pat drew card 7 Sam drew card 13 Sam has high card Pat: 4 Sam: 14  
Pat drew card 10 Sam drew card 6 Pat has high card Pat: 6 Sam: 14  
Pat drew card 9 Sam drew card 7 Pat has high card Pat: 8 Sam: 14  
Pat drew card 4 Sam drew card 13 Sam has high card Pat: 8 Sam: 16  
Pat drew card 3 Sam drew card 3 War Pat: 8 Sam: 16  
Pat drew card 11 Sam drew card 3 Pat has high card Pat wins war of 4 cards Pat: 12 Sam: 16  
Pat drew card 4 Sam drew card 10 Sam has high card Pat: 12 Sam: 18  
Pat drew card 12 Sam drew card 11 Pat has high card Pat: 14 Sam: 18  
Pat drew card 4 Sam drew card 11 Sam has high card Pat: 14 Sam: 20  
Pat drew card 8 Sam drew card 5 Pat has high card Pat: 16 Sam: 20  
Pat drew card 12 Sam drew card 9 Pat has high card Pat: 18 Sam: 20  
Pat drew card 5 Sam drew card 6 Sam has high card Pat: 18 Sam: 22  
Pat drew card 10 Sam drew card 13 Sam has high card Pat: 18 Sam: 24  
Pat drew card 2 Sam drew card 2 War Pat: 18 Sam: 24  
Pat drew card 14 Sam drew card 14 War Pat: 18 Sam: 24  
Pat drew card 2 Sam drew card 5 Sam has high card Sam wins war of 6 cards Pat: 18 Sam: 30  
Pat drew card 11 Sam drew card 14 Sam has high card Pat: 18 Sam: 32  
Pat drew card 10 Sam drew card 3 Pat has high card Pat: 20 Sam: 32  
Final Score Pat: 20 Sam: 32 Winner: Sam

### 1.2.1 Deck Shuffling

While seemingly simple—shuffling a deck is a somewhat complicated problem. Luckily, Python's random library has a built in shuffle algorithm. Feel free to read the documentation, but we have provided a simple wrapper function that will return to you a shuffled deck of cards.

```
import random

name: shuffled_deck
purpose: will return a shuffled deck to the user
input:
returns: a list representing a shuffled deck
def shuffled_deck():
 basic_deck = list(range(2, 15)) * 4
 random.shuffle(basic_deck)
 return basic_deck
```



### 1.2.2 Bonus

Instead of closing the program when the deck is empty, create a way for the user to play again.

*formatted by Markdeep 1.093 ↗*

# TEALS Program

[Home](#) | [Curriculum Map](#) | [Additional Readings](#) | [Change Log](#)

## 1 Lesson 3.04: Debugging and Scope

---

### 1.1 Learning Objectives

---

Students will be able to...

- Define and identify scope, aliasing, stack trace
- Demonstrate changing a list in a function updates the list outside of the function
- Demonstrate updating variables in a function does not affect the variable outside of the function
- Demonstrate using global variables
- Draw a simple stack trace

### 1.2 Materials/Preparation

---

- [Do Now](#)
- [Lab - Aliasing & Scope \(printable lab document\)](#) ([editable lab document](#))
- Read through the do now, lesson, and lab so that you are familiar with the requirements and can assist students.
- [Associated Readings 3.4](#)

### 1.3 Pacing Guide

---

Duration	Description
5 Minutes	Do Now
10 Minutes	Lesson
30 Minutes	Lab/Review
10 Minutes	Debrief

### 1.4 Instructor's Notes

---

#### 1.4.1 1. Do Now

- Students have a chance to think about what & discuss what concepts they have been most challenged by.

- Next, students practice passing a list as an argument and updating that list within the function.

## 1.4.2 2. Lesson

- Discuss what students observed in the Do Now and take time, if needed, to go over questions about concepts that students find challenging.

### 1.4.2.1 Aliasing

- Explain the concept of **aliasing**.
- You can draw on the board a diagram of the variable pointing to a list.
- Note that when passing the location of a list you are not passing the actual value, so the list can be changed.
- **Video Explanation of Aliasing**

[

]([https://www.youtube.com/watch?v=7m\\_cw3otyro](https://www.youtube.com/watch?v=7m_cw3otyro))

*Python - aliasing*

### 1.4.2.2 Scope of functions

- Explain to students that variable scope is the part of a program where a variable is accessible
- A variable which is defined in the main body of a file is called a global variable.

- **Video explanation of Variable Scope**

[

](<https://youtu.be/Ao54Ged9suI>)

*Python - Scope*

### 1.4.2.3 Global variables: variables defined outside of a function and used in many different functions

- To modify global variables defined outside the function you must declare the variable with the statement `global name_of_variable`.
- Any variable created inside of a function is a local variable.
- Variables in functions include the function parameters, the variables defined in the function and variables declared as global.
- Local variables of functions can't be accessed from outside when the function call has finished.
- Explain global variables are often used for constants.
- NOTE: We use the 'ALL CAPS' convention for global variables.

### 1.4.2.4 Conventions

- Discuss that programming languages frequently have **conventions**. It helps make code more readable, but isn't essential to functionality (in most cases). Here are some examples,
- camelCase for function names
- ALL\_CAPS for global variables

underscore\_separated for variables

### 1.4.2.5 Stick Diagrams

- Demonstrate how to draw the Stack Diagrams shown in the course book ([found in section 3.4](#)) and explain how they show the scope of variables as they relate to functions.
- Point out the error messages that will occur if you use a variable out of its scope.

#### 1.4.2.6 Debugging

- Help students follow their program to understand how the code is working
- Explain how the use of print statements throughout your code can let you know where in the program things are not operating as expected.

#### 1.4.3 3. Lab

- This lab has students running code that gets them thinking about aliasing and scope. They must also create a stack trace for a program to show their understanding of scope.

#### 1.4.4 4. Debrief

- Take time to review the concepts covered today: **scope**, **aliasing**, and **stack traces**.
- Call a few students to the board to draw their stack traces from the lab and talk through them.

### 1.5 Accommodation/Differentiation

---

If students are moving quickly, they can look ahead at the project spec or research the game Oregon Trail for context.

### 1.6 Forum discussion

---

[Lesson 3.04: Debugging and Scope \(TEALS Discourse Account Required\)](#)

formatted by [Markdeep 1.093](#) 

# TEALS Program

[Home](#) | [Curriculum Map](#) | [Additional Readings](#) | [Change Log](#)

## 1 Do Now 3.04

---

### 1.1 Quiz Announcement

---

We will have a quiz next class covering all of the above topics.

#### 1.1.1 Discussion

Is there any topic you would like to focus on and cover more of?

#### 1.1.2 In your Notebook

Rank the following from easiest to hardest:

1. Importing built-in functions
2. Using randint
3. Abstraction/creating functions
4. Passing int/str/float/bool arguments into functions
5. Calling a function
6. List syntax
7. Return vs print

#### 1.1.3 Type the following into the console

```
my_list = ['a', 'b', 'c', 'd']
input: a list of strings
output: None
def my_function(list_argument):
 list_argument[0] = 'z'
print(my_list)
my_function(my_list)
print(my_list)
```



1. What did the program output and what is this program doing?

#### 1.1.4 Bonus

Try writing a similar program but passing in integers instead of a list. What happens?



# TEALS Program

[Home](#) | [Curriculum Map](#) | [Additional Readings](#) | [Change Log](#)

## 1 Lab 3.04 - Aliasing & Scope

---

### 1.1 In Your Notebook

---

#### 1.1.1 Aliasing

1. Will updating b affect a? Explain why or why not?

```
a = [1, 2, 4]
b = a
```



2. Predict what my\_list list will print out when this code is run. If you are not sure check the code by copying and running it.

```
input: a list of ints
output: an int
def update_list(a_list):
 a_list[3] = "yo"
 b = a_list[4]
 b = 100

my_list = [1, 2, 3, 4, 5]
update_list(my_list)
```



#### 1.1.2 Scope

1. Draw a stack diagram for the following:

```
var_1 = "kittens"
var_2 = "cookies"

input: a string
output: a string
def my_function(my_favorite_things):
 song_lyrics = "rain drops on roses,"
 combined_song = song_lyrics + my_favorite_things
 return combined_song

input: a string
output: a string
def my_function_2(item, item2):
 full_lyrics = item + "on " + item2
 full_song = my_function(full_lyrics)
 return full_song

my_song = my_function_2(var_1, var_2)
```



### 1.2 Complete the following on your own

---

1. Write down what (if anything) is wrong with the following code.
2. If there was an issue write out how to fix it.
3. If you are unsure copy and run the code and fix it

### 1.2.1 Problem 1

```
var_1 = 'cat'
var_2 = 'dog'

def print_out_my_favorite(favorite_pet):
 if favorite_pet == var_1:
 print("My favorite pet is the cat.")
 if favorite_pet == var_2:
 print("My favorite pet is the dog.")
 var_2 = "cat"

print_out_my_favorite(var_1)
print(var_2)
```



### 1.2.2 Problem 2

```
var_1 = 'cat'
var_2 = 'dog'

def print_out_my_favorite(favorite_pet):
 var_1 = 'dog'
 var_2 = 'cat'
 if favorite_pet == var_1:
 print("My favorite pet is the cat.")
 if favorite_pet == var_2:
 print("My favorite pet is the dog.")

print_out_my_favorite(var_1)
print(var_1 + " " + var_2)
```



### 1.2.3 Problem 3

```
var_1 = 'cat'
var_2 = 'dog'

def print_out_my_favorite(favorite_pet):
 if favorite_pet == var_1:
 print("My favorite pet is the cat.")
 if favorite_pet == var_2:
 print("My favorite pet is the dog.")

print_out_my_favorite(var_1)
print(var_2)
```



## 1.3 In your console

---

### 1.3.1 Write a program using the following specifications

1. That has a global variable, `my_num`.
2. Create three functions that update `my_num`
3. `add2`: this function adds 2 to `my_num`
4. `multiply_num`: this function takes in a parameter, `multiplier`, and multiplies `my_num` by that parameter
5. `add2_and_multiply`: this function takes in a parameter, `multiplier`, and calls `add2`, then calls `multiply_num`.

### 1.3.2 Complete the program

Write the following code in the main part of the program.

1. sets `my_num` to some initial value you choose
2. prints `my_num`
3. calls `add2_and_multiply()` with some argument you choose
4. prints the final value of `my_num`
5. Confirm that the printed values match what you expected.

*formatted by [Markdeep 1.093](#)* 

# TEALS Program

[Home](#) | [Curriculum Map](#) | [Additional Readings](#) | [Change Log](#)

## 1 Lesson 3.05: Project 3

---

### 1.1 Learning Objectives

---

Students will be able to... \* Use project planning skills to complete a longer-term project \* Create functions to organize a project \* Apply skills learned in units 1-3 to create a functioning program

### 1.2 Materials/Preparation

---

- Project Spec - Oregon Trail ([printable project Spec](#)) ([editable project spec](#))
- Project Spec - Crosscountry Canada
- Project Spec - Daily Planner ([printable alternate project Spec](#)) ([editable alternate project spec](#))
- [Oregon Trail Starter Code](#)
- Solution (access protected resources by clicking on "Additional Curriculum Materials" on the [TEALS Dashboard](#))
- Update the Project Spec as needed to meet your grading requirements
- Try creating your own variation on the Oregon Trail code so you are familiar with the potential challenges and bugs your students will hit.
- Review [4 Steps to Solve Any CS Problem](#)
- [Editable Grading Rubric](#)

#### 1.2.1 Day 1

Duration	Description
10 Minutes	Project Overview
40 Minutes	Planning
5 Minutes	Debrief

#### 1.2.1.1 Days 2-9

Duration	Description
5 Minutes	Review Day Plan
45 Minutes	Project Work
5 Minutes	Debrief

## 1.3 Instructor's Notes

---

### 1.3.1 1. 4 Steps to Solve Any CS Problem

- Remind students of the [4 Steps to Solve Any CS Problem]

### 1.3.2 2. Project Overview

- Demo the Oregon Trail finished project.
- Give out the project spec and go over game rules.

### 1.3.3 3. Planning

1. Have students draw out the game play
2. Students should plan to create functions for each user interaction by figuring out where the repeated code will be.
3. Students should list out which variables they will need.
4. Have students plan out their next 7 days. Suggested timeline/checkpoints below:
  - Set up user inputs with dummy functions, make sure game loop works
  - Create variables necessary to run the game, start implementing basic functions
  - Focus on the random functions
  - Figure out how to move the days
  - Finish day updating
  - Connect functions together
  - Wrap up and game over check is correct

## 1.4 Accommodation/Differentiation

---

- Advanced students can add in random events like cholera or snake bites.
- Students can also have a list of travelers instead of just 1, where each traveler is affected differently by each action.
- The planning phase of this project will be essential,
- especially for students who you think may struggle with this project.
- Provide more guidance and scaffolding to those students that need it.

## 1.5 Grading

---

### 1.5.1 Objective Scoring Breakdown

Points	Percentage	Objective	Lesson
3	10%	Student correctly identifies data types	2.01
3	10%	Student correctly uses lists	2.04, 2.05
3	10%	Student correctly uses built in functions	3.01
12	38%	Student can program using user-defined functions	3.02, 3.03, 3.04
5	16%	Student can decompose a problem to create a program from a brief	
5	16%	Student uses naming/ syntax conventions and comments to increase readability	
31		<b>Total Points</b>	

### 1.5.2 Scoring Consideration

You may need to adjust the points in order to fit your class. Treat the percentages as a guide to determine how to weight the objectives being assessed.

## 1.6 Forum discussion

---

Lesson 3.05: Oregon Trail (TEALS Discourse Account Required)

formatted by Markdeep 1.093 

# TEALS Program

[Home](#) | [Curriculum Map](#) | [Additional Readings](#) | [Change Log](#)

## 1 Project 3: Oregon Trail

---

Using variables, functions, and conditionals in Python, students will create an Oregon Trail game.

### 1.1 Overview

---

We will be recreating Oregon Trail! The goal is to travel from Independence, Missouri to Oregon City, Oregon (2000 miles) by Dec 31st. However, the trail is arduous. Each day costs you food. You can hunt and rest, but you have to get there before winter!

### 1.2 Details

---

#### 1.2.1 Behavior

- Player starts in Independence, MO on 03/01 with 2,000 miles to go, 500lbs of food, and 5 health.
- The player must get to Oregon by 12/31
- At the beginning of the game, user is asked their name.
- Each turn, the player is asked what action they choose, where the player can type in the following: travel, rest, hunt, status, help, quit
- The players health will decrease twice each month.
- The player eats 5lbs of food a day.
- travel: moves you randomly between 30-60 miles and takes 3-7 days (random).
- rest: increases health 1 level (up to 5 maximum) and takes 2-5 days (random).
- hunt: adds 100lbs of food and takes 2-5 days (random).
- status: lists food, health, distance traveled, and day.
- help: lists all the commands.
- quit: will end the game.

#### 1.2.2 Implementation details

- Create functions for all options a player can take
- Use global variables to keep track of player health, food pounds, miles to go, current day, current month
- Create a function add\_day which updates the day
- Use a list to keep track of which months have 31 days and use this in the add\_day function (i.e.: MONTHS\_WITH\_31\_DAYS = [1, 3, 5, 7, 8, 10, 12])
- Create a function select\_action which uses a while loop to call add\_day function
- Game ends if days run out, health runs out, you get there (Oregon), or the player quits.

### 1.3 Bonus

---

1. Make the rate of food consumption be a function of activity. So if a player hunts for a turn they take up more food, but if they rest they take up less food.

2. Create a random event that occurs randomly once a month, like a river crossing or a dysentery, that will take up a range of 1-10 food, 1-10 days and 0-1 health.

formatted by Markdeep 1.093 

# TEALS Program

[Home](#) | [Curriculum Map](#) | [Additional Readings](#) | [Change Log](#)

## 1 Alternate Project 3: Daily Planner

---

Created by Brian Weinfeld

Using variables, functions, and conditionals in Python, you will create a daily planner program.

### 1.1 Overview

---

In this project, you will be creating a daily planner program. A daily planner helps a user track what they need to do today and when they need to do it. This can be thought of as a TODO list that pairs each action with a time when the action is done. A planner can help make sure a person doesn't accidentally book two things for the same time. Do you add events to your phone's calendar?

### 1.2 Details

---

#### 1.2.1 Behavior

Welcome to your Daily Planner

What would you like to do? (add, clear, display, exit) add

What event would you like to add? go to school

What time? 8

Is this event critical? no

Added go to school at 8

[', ', ', ', '']

[', ', ', ', '']

['go to school', '', '', '']

[', ', ', ', '']

[', ', ', ', '']

[', ', ', ', '']

What would you like to do? (add, clear, display, exit) add

What event would you like to add? wake up

What time? 6

Is this event critical? no

Added wake up at 6

[', ', ', ', '']

[', ', 'wake up', '', '']

['go to school', '', '', '']

[', ', ', ', '']

[', ', ', ', '']

[', ', ', ', '']

What would you like to do? (add, clear, display, exit) add

What event would you like to add? do homework

What time? 6

Is this event critical? no

Conflict with wake up

What would you like to do? (add, clear, display, exit) display

[', ', ', ', '']

[', ', 'wake up', '']

```
[['go to school', '', '', '']
[', , , ,]
[, , , ,]
[, , , ,]]

What would you like to do? (add, clear, display, exit) add
What event would you like to add? eat breakfast
What time? 6
Is this event critical? yes
Replacing wake up at 6 with eat breakfast
[, , , ,]
[, , 'eat breakfast', ,]
['go to school', , , ,]
[, , , ,]
[, , , ,]
[, , , ,]

What would you like to do? (add, clear, display, exit) clear
What time to start clearing? 5
What time to end clearing? 7
Clearing [, 'eat breakfast']
[, , , ,]
[, , , ,]
['go to school', , , ,]
[, , , ,]
[, , , ,]
[, , , ,]

What would you like to do? (add, clear, display, exit) exit
Goodbye
```

### 1.2.2 Implementation Details

The planner is a list that has 24 empty strings in it. (The empty string is ""). Each of the 24 indexes represent 1 hour of time in a day. For example, index 0 represents 12:00AM to 12:59AM, index 1 represents 01:00AM to 01:59AM and index 23 represents 11:00PM to 11:59PM. Your program will allow a user to add events at certain times of the day and prevent conflicts from occurring. All of the events will take exactly 1 hour.

Think back to the previous project where you created a TODO list program. All of the code in that program was put into a single script. This is OK when the program is small, but it gets harder and harder to code as the project gets bigger. Did you find the end of that project to be more challenging than the beginning? You can help mitigate this problem by creating functions to handle large amounts of work.

Begin this project by creating your 24 length list of empty strings and the following three functions. Do your best to ensure that these functions work properly before moving onto the rest of the project.

```
def add_event(planner, event, time, critical):
 # finish function
```

- `add_event` takes the planner (your list), the event you wish to add (a string), the time you want to do the event (an int between 0 to 23 inclusive) and whether the event is critical (a Boolean). If there is no event in your planner for the indicated time, add it to the planner. If there is already an event for that time you have a conflict! (You can't do both things at the same time.) Use `critical` to figure out what to do. If `critical` is True, then the new event takes priority. Replace the event in the planner at that time with the new event. If `critical` is False, the event is not more important and instead an error should be printed. No change is made to the list. Return True if a new event is added to the planner and False if no changes were made.

```
def display_planner(planner):
 # finish function
```

- `display_planner` takes the planner (your list) and displays it in an easy to read fashion. Print 6 lines, each line containing 4 events. So the first line will contain events from the first four indexes, the second line will contain the next four and so on.

```
def clear_timeframe(planner, timeframe):
 # finish function
```

- `clear_timeframe` takes the planner (your list) and a timeframe (slice) that you wish to clear. Sometimes you need to remove events that have been cancelled. Clear out all of the events that occur during the given timeframe. Be careful, a slice can contain multiple elements. You need to set all the events in the slice back to the empty string.
- Once you have finished the above functions, create a script that allows a user to add and remove elements from their planner. The user should have four choices **add**, **clear**, **display**, **exit**. **add** is used to add events to the planner, **clear** is used to remove events from the planner, **display** is used to display the day's events and **exit** is used to exit the program. You should also automatically print the planner after any changes have been made to it. See the below running example for details.

### 1.2.3 Challenge

This section contains additional components you can add to the project. These should only be attempted after the project has been completed.

- Some events take longer than one hour. Modify the `add_event` function so that time is now a slice representing the beginning and ending time of the event. You will need to place the event in each index in your planner. Be careful, the new event can only be placed if either the event is critical (overwriting any event already in the planner at that time) or every hour in the planner is empty.
- Combine all the information into one line so that it is easier to interact with the planner. For example, you might add a new event

by entering “add go to school 8 13 critical” or “add practice lines 14 15 not”

### 1.2.4 Super Challenge

The super challenge will require knowledge that has not been taught yet. You will need to do additional research on your own. Good luck!

Sometimes the time the event is scheduled for is less important than making sure the event is on the schedule. Add a parameter to the functions that add an element to the schedule called **force**. If force is true, the event must be scheduled. If it can't be scheduled at the expected time, find the closest available timeslot in the future and schedule it there instead. If the event is critical, then the displaced event should be moved forward. Be sure to alert the user that the event has been scheduled, but at a different time than expected! The placement will still fail if there are no available timeslots to place the event.

# TEALS Program

[Home](#) | [Curriculum Map](#) | [Additional Readings](#) | [Change Log](#)

## 1 Unit 4 - For Loops and Nesting

---

### 1.1 Essential Questions

---

- How do you iterate through the items of a list using a for loop?
- How do you use a nested for loop to do a stack trace?
- How do you use for loops to traverse a list?
- How do you debug programs utilizing nested for loops and lists?"

### 1.2 Pacing Guide

---

#### 1.2.1 Timing

1 Day = 50 minute class period

Lesson	Days	Resources
4.01: Looping Basics	1	<a href="#">4.01 Slide Deck</a>
4.02: For Loops, Nested Loops	1	<a href="#">4.02 Slide Deck</a>
4.03: Nested For Loops	2	<a href="#">4.03 Slide Deck</a>
4.04: Nested Lists & Looping	2	<a href="#">4.04 Slide Deck</a>
4.05: Debugging and Quiz	1	<a href="#">4.05 Slide Deck</a>
4.06: Tic-Tac-Toe	9	<a href="#">4.06 Slide Deck</a>
<b>Total Days</b>	16	
<b>Total Minutes</b>	800	

### 1.3 Key Terms

---

- For Loop
- Item
- Iteration
- Scope

- Range
- Nested For Loops
- Stack Trace
- Nested List

formatted by Markdeep 1.093 

# TEALS Program

[Home](#) | [Curriculum Map](#) | [Additional Readings](#) | [Change Log](#)

## 1 Lesson 4.01: Looping Basics

---

### 1.1 Learning Objectives

---

Students will be able to...

- Define and identify: **for loop, item, iteration, scope**
- Recall looping in Snap! and reapply the concept in Python
- Loop through (traverse) the items in a list
- Be aware of the scope of variables during iteration

### 1.2 Materials/Preparation

---

- [Do Now](#)
- [Lab - de\\_vowel \(printable lab document\)](#) ([editable lab document](#))
- [Associated Reading](#)
- Read through the do now, lesson, and lab so that you are familiar with the requirements and can assist students

### 1.3 Pacing Guide

---

Duration	Description
10 Minutes	Do Now
10 Minutes	Lesson
30 Minutes	Lab
5 Minutes	Debrief

### 1.4 Instructor's Notes

---

#### 1.4.1 1. Do Now

- Display the Do Now on the board.
- Students will experience how to use a for loop to efficiently complete a repeated action.

## 1.4.2 2. Lesson

- Go over Part 1 of the Do Now. Ask the students what would happen if the list got much larger?
- If they say they would write down a lot of code, ask how readable that might be, or how long would it take to write, or the greater potential for bugs due to typos.
- Go over part 2 of the Do Now. Ask the students what happened. Ask if they remember something similar from Snap!
- Introduce the **for loop** as a way to deal with issues associated with part 1 of the Do Now.

### 1.4.2.1 Video Explanation of For Loops

[

](<https://youtu.be/KosrKNJK9Sw>)

*For Loops Explanation*

- From the Code in the Do Now: `for num in list_of_numbers:`
- Emphasize that the body of the for loop is the indented part
- **Iteration:** body of the loop is repeated with different values of the list. Note how the body of the loop is repeated but `num` changes. Consider drawing this out on the board.
- Remind students of the concept of **scope**, showing how `num` changes values with each iteration of the loop.
- Go over Part 3 (many students likely didn't have time to finish). Ask students to write the first line of the loop on the board. Have students brainstorm what the body should be. Come to a group consensus and run the code.

## 1.4.3 3. Lab

- De-vowel lab: Students will create a function that will take in a sentence and return that sentence without vowels.

## 1.4.4 4. Debrief

- Talk about any issues or challenges the students had with this lab. If there is time, call students up to the board to show and demonstrate their code/solutions.

## 1.5 Accommodation/Differentiation

---

If there is time, go over the bonus question. Explain how a counter is a frequently used tool to keep track of the count of things from outside the loop . Discuss the concept of the counter's scope (counter gets updated in the loop, but doesn't reset automatically at each iteration). Counters can be used with any loop, and are often used with while loops!

## 1.6 Forum discussion

---

[Lesson 4.01: Looping Basics \(TEALS Discourse Account Required\)](#)



# TEALS Program

[Home](#) | [Curriculum Map](#) | [Additional Readings](#) | [Change Log](#)

## 1 Do Now 4.01

---

### 1.1 1. In your Console

---

#### 1.1.1 Type the following Code

```
single_fruit = ['apple', 'banana', 'watermelon', 'grape']
multi_fruit = []
multi_fruit.append(single_fruit[0] + 's')
multi_fruit.append(single_fruit[1] + 's')
multi_fruit.append(single_fruit[2] + 's')
multi_fruit.append(single_fruit[3] + 's')
print(multi_fruit)
```



### 1.2 In your Notebook

---

#### 1.2.1 Respond to the following

Briefly write down what happened. What would happen if you added 100 items to the list `single_fruit`? Write down how you would update `multi_fruit`.

### 1.3 2. In your Console

---

#### 1.3.1 Type the following

```
list_of_numbers = [3, 5, 10, 23]
for num in list_of_numbers:
 print("num is " + str(num))
```



### 1.4 Continue in your Notebook

---

#### 1.4.1 Responses to the following

Briefly write down what happened. How would this change if you added 100 items to `list_of_numbers`?

## **1.5 3. In your Console**

---

### **1.6 Rewrite the code from part 1 using knowledge from part 2**

---

*formatted by [Markdeep 1.093](#) *



# TEALS Program

[Home](#) | [Curriculum Map](#) | [Additional Readings](#) | [Change Log](#)

## 1 Lesson 4.02: For Loops Using Range

---

### 1.1 Learning Objectives

---

Students will be able to...

- Define and identify: **range**
- Use the range and len function to update lists via for loops

### 1.2 Materials/Preparation

---

- [Do Now](#)
- [Lab - Getting Loopy \(printable lab document\)](#) ([editable lab document](#))
- [Associated Reading](#)
- Read through the Do Now, lesson, and lab so that you are familiar with the requirements and can assist students

### 1.3 Pacing Guide

---

Duration	Description
5 Minutes	Do Now
10 Minutes	Lesson
35 Minutes	Lab
5 Minutes	Debrief

### 1.4 Instructor's Notes

---

#### 1.4.1 1. Do Now

- Display the Do Now on the board.
- Students experiment with and are introduced to the range function.

## 1.4.2 2. Lesson

### 1.4.2.1 Part 1 of the Do Now

- Ask students what the **range** function did.
- Remind them that there are reference docs online.
- Show the docs for the range function (note that it actually takes in a third value that is optional).
- Work together with the students to write a for loop just using the range function.

### 1.4.2.2 Part 2 of the Do Now

- Ask the students what happened
- Ask the students why these values might be helpful
- They are a list of the indices!

### 1.4.2.3 Part 3 (many students likely didn't finish)

- Ask students to write the first line of the loop on the board
- Work together as a class to come to a solution that is demonstrated for all to see.

## 1.4.3 3. Lab

- Students re-write the fruit pluralize program from yesterday's do now, but without creating a new list.
- Students write a function that reverses the letters in a string.

## 1.4.4 4. Debrief

- Talk about any issues the students had with the lab today.
- Discuss how lists are mutable, so you don't have to return a new value. Instead, the list is just updated as the loop runs.

### 1.4.4.1 Video explanation on Python Lists being Mutable

[

]([https://youtu.be/\\_y3PqL4lIzw?t=181](https://youtu.be/_y3PqL4lIzw?t=181))

*Mutable Python Lists*

## 1.5 Accommodation/Differentiation

---

- If students are having a hard time with the fruit pluralize, consider altering the fruit program to not allow fruit that ends in y.
- Some students may have issues with grabbing the last item of a string, consider providing tips or scaffolding for students that are struggling with this.
- Go over the bonus question if any students got to it. Discuss having a function inside of the loop and how that operated.

## 1.6 Forum discussion

---

*formatted by Markdeep\_1.093* ↗

# TEALS Program

[Home](#) | [Curriculum Map](#) | [Additional Readings](#) | [Change Log](#)

## 1 Do Now 4.02

---

### 1.1 1. In your Console

---

#### 1.1.1 Type the following code

```
for i in range(0, 10):
 print(i)
```



### 1.2 In your Notebook

---

#### 1.2.1 Respond to the following

Write down what the range function does.

### 1.3 2. In your Console

---

#### 1.3.1 Type the following

Use the range and len functions to make a for loop that loops through a.

```
a = ['apples', 'oranges', 'pears', 'grapes']

len_a = len(a)
range(0, len(a))
```



### 1.4 Continue In your Notebook

---

1. Write down what range(0, len(a)) does.
2. What is the return value of the range function?

### 1.5 3. In your Console

---

Use the range and len functions to make a for loop. (Remember that for loops iterate over lists)



# TEALS Program

[Home](#) | [Curriculum Map](#) | [Additional Readings](#) | [Change Log](#)

## 1 Lab 4.02 - Getting Loopy

---

### 1.1 Part 1

---

Write a function `pluralize_words` that takes in a list of words and updates the values of the list to make each one plural. It returns nothing. Making plurals in English has a number of special cases, but for this lab we'll use a simple rule: if the word ends in a `y` remove the `y` and add `ies`; otherwise add an '`s`'.

We'll exercise the function on a list of words.

1. Create the function contract for `pluralize_words`.
2. Provide a few examples that confirm `pluralize_words` works as expected:
  - Include examples with 'berry'
  - What if the list is empty?
  - What happens if a word ends in 's'?

#### 1.1.1 Example 1

```
contract goes here
def pluralize_words(word_list):
 # your code goes here

word_list = ['apple', 'berry', 'melon']
print("Singular words: " + str(word_list))
pluralize_words(word_list)
print("No longer singular words: " + str(word_list))
 # more examples go here
```

##### 1.1.1.1 Here is what it should look like when you run your code

```
Singular words: ['apple', 'berry', 'melon']
No longer singular words: ['apples', 'berries', 'melons']
```

#### 1.1.2 Hint

Remember that you can index into the string and get the length of a string. Use that to get the last letter of each word.

### 1.2 Part 2

---

Create a function `my_reverse`, which will return a reversed string.

1. Create the function contract for `my_reverse`.

2. Provide a few examples to confirm that `my_reverse` works:

- o An empty string
- o A string of even length
- o A string of odd length greater than 1
- o A string of length 1

### 1.2.1 Example 2

```
contract goes here
def my_reverse(string_to_reverse):
 # your code goes here
reversed = my_reverse("apples")
print(reversed)
examples go here
```

### 1.2.2 Here is what example 2 should look like when you run your code

```
>>> python3 my_reverse_lab.py
selppa
```

### 1.2.3 Hint 2

To get the last element: `(len(my_list) - 1)` - 0 To get the second to last element: `(len(my_list)-1 ) - 1` To get the third to last element: `(len(my_list)-1) - 2`

### 1.2.4 Bonus

Create a function `reverse_strings_in_list`. This function will input a list of strings you want to reverse. The function will reverse the strings in the list by calling the `my_reverse` function in a loop.

formatted by [Markdeep 1.093](#) 

# TEALS Program

[Home](#) | [Curriculum Map](#) | [Additional Readings](#) | [Change Log](#)

## 1 Lesson 4.03: Nested For Loops

---

### 1.1 Learning Objectives

---

Students will be able to...

- Define and identify: **nested for loops, stack trace**
- Use nested for loops via a function and a for loop
- Use nested for loops via two loops nested
- Use a stack trace to understand and demonstrate the flow of nested for loops

### 1.2 Materials/Preparation

---

- Do Now
- [Lab - Nested For Loops \(printable lab document\)](#) ([editable lab document](#))
- Read through the Do Now, lesson, and lab so that you are familiar with the requirements and can assist students
- Video Explanation of Nested For Loops

[

](<https://youtu.be/fyP4SXpkYG4>)

*Nested For Loops*

#### 1.2.1 Day 1 Pacing

Duration	Description
5 Minutes	Do Now
10 Minutes	Lesson
35 Minutes	Lab
5 Minutes	Debrief

#### 1.2.2 Day 2 Pacing

Duration	Description
5 Minutes	Do Now
10 Minutes	Review
35 Minutes	Lab
5 Minutes	Debrief

## 1.3 Instructor's Notes

---

### 1.3.1 1. Do Now

- Display the Do Now on the board.
- Students use nested for loops to create a square star pattern.

### 1.3.2 Lesson

#### 1.3.2.1 Go over part 1 of the Do Now

- Discuss the output of the program - were the students able to guess the output without typing it?
- Go over how to read for loops if students are struggling (drawing the [loop diagram]).
- Make sure students are understanding loops and string concatenation.
- If students continue to struggle, take 5 minutes to go over the loop syntax and practice.

#### 1.3.2.2 Go over part 2 of the Do Now

- Ask a student to write the `print_star_square` function on the board.
- Define **nested for loop**: a loop within another loop.
- For each iteration of the outer loop the inner loop is iterated through completely.
- Draw a diagram (**stack trace**) of the for loop (e.g. something like [loop diagram])
- Ask students to draw the nested part of the state diagram (should be inside the outer loop but look the same as the outer loop)

#### 1.3.2.3 Go over part 3 of the Do Now

- If students were unable to finish this, give them 5 minutes to practice in groups before calling them back to go over this part.
- Ask a couple students to write on the board how they did this
- Ask them how treating the loop as its own function made it easier or harder.
- Ideally this should make it easier as a way of abstracting knowledge of looping.

### 1.3.3 3. Lab

- The lab asks students to write functions that produce different outputs using nested for loops.

#### 1.3.4 4. Debrief

- Inform students that there will be a Unit 4 Quiz after Lesson 4.04.
- Go over common questions the students had.
- On the second day, if time allows, go over the bonus and discuss how students solved the problem.

#### 1.3.5 Accommodation/Differentiation

- If students need extra time for lab there is another day in the schedule for that.
- This topic is often confusing for students new to the concept, so build in time for frequent individual checks for understanding.
- **Bonus Lab Problem** This is a bit more difficult and should allow students who are moving quickly a challenge

### 1.4 Forums discussion

---

Lesson 4.03: Nested For Loops (TEALS Discourse Account Required)

formatted by Markdeep 1.093 

# TEALS Program

[Home](#) | [Curriculum Map](#) | [Additional Readings](#) | [Change Log](#)

## 1 Do Now 4.03

---

### 1.1 Type in your Console

---

```
def print_6_stars():
 my_string = ''
 for i in range(0, 6):
 my_string += '*'
 print(my_string)
```



### 1.2 In your Notebook

---

1. Write down what the output of the function `print_6_stars` is.
2. Write a function `print_star_square` that calls `print_6_stars` in a loop to produce the following output.

```
>>>python3 print_stars.py
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
```



3. Rewrite the function `print_star_square` without using `print_6_stars`.
  - o Note that there are two ways to get the above output, so just try doing both!

formatted by [Markdeep 1.093](#) 





# TEALS Program

[Home](#) | [Curriculum Map](#) | [Additional Readings](#) | [Change Log](#)

## 1 Lesson 4.04: Nested Lists & Looping

---

### 1.1 Learning Objectives

---

Students will be able to...

- Define and identify: **nested list**
- Use nested for loops to traverse through nested lists

### 1.2 Materials/Preparation

---

- [Do Now](#)
- [Lab - Shopping List \(printable lab document\)](#) ([editable lab document](#))
- Read through the Do Now, lesson, and lab so that you are familiar with the requirements and can assist students
- Explanation video of Nested Lists

[

](<https://youtu.be/kzZR9veV78A>)

*Nested List*

#### 1.2.1 Day 1 Pacing

Duration	Description
5 Minutes	Do Now (Part 1)
10 Minutes	Lesson
35 Minutes	Lab (Part 1)
5 Minutes	Debrief

#### 1.2.2 Day 2 Pacing

Duration	Description
5 Minutes	Do Now (Part 2)
10 Minutes	Lesson
35 Minutes	Lab (Part 2)
5 Minutes	Debrief

## 1.3 Instructor's Notes

---

### 1.3.1 1. Do Now (Day 1)

- Display the Do Now on the board.
- Make sure students only work on the first part of the Do Now, as the second part will be for the next lesson.

### 1.3.2 2. Lesson (Day 1)

#### 1.3.2.1 Go over the first problem of the Do Now

- Ask students what type `my_building` is.
- Discuss **nested lists** as lists that have each element as a list. Ask students to think of some other data that might fit into a **nested list**. What about a game of Tic-Tac-Toe or chess?
- Go into depth a bit more on nested lists.
- Ask the students how they would access different parts of the `my_building` example.
- Write down the syntax of `[][]`.
- In the `my_building` example the first bracket gets you the floor and the second gets you the room.

#### 1.3.2.2 Practice

- Have students practice writing their own nested lists, or work in pairs with one student writing one nested lists and the other student writing another.

#### 1.3.2.3 Go over the second problem in the Do Now

- Asks the students what happened when they iterated over `my_building`.
- As an extension, ask the students how they would print out each `b` apartment.

### 1.3.3 3. Lab (Day 1)

- Students will practice accessing items from lists of lists by creating a schedule program and accessing/updating elements.

#### **1.3.4 4. Debrief (Day 1)**

- Make sure that all students are able to access elements from a list of lists.

#### **1.3.5 5. Lesson (Day 2)**

##### **1.3.5.1 Go over the 1st problem of the Do Now**

- This should be a review of looping.
  - If students are having trouble, take extra time to review looping syntax and procedures.

##### **1.3.5.2 Go over the 2nd problem of the Do Now**

- Ask students to write on the board what they did. Discuss how you would write this without the extra function.
- Go over how to write this program, using student responses.
- As extensions to this discussion, have students consider how they would alter the program to not print out the middle floor. Or how would they alter it to not print out any apartment a's?

#### **1.3.6 6. Lab (Day 2)**

- Students will create functions that loop through lists of lists.

#### **1.3.7 7. Debrief (Day 2)**

- Ask students if there was any difficulty looping through lists of lists.
- Remind students there will be a quiz next class covering everything up to (and including) nested lists.

### **1.4 Accommodation/Differentiation**

---

Students may struggle with the concept of nested lists. Be prepared to have additional analogies to help students understand the usefulness of the concept.

- Ex. What if you keep a weekly to-do list? The days of the week would be the first list, and then each item that you hope to accomplish under each day is the list within the list.

### **1.5 Forum discussion**

---

[Lesson 4.04: Nested Lists & Looping](#)

# TEALS Program

[Home](#) | [Curriculum Map](#) | [Additional Readings](#) | [Change Log](#)

## 1 Do Now 4.04

---

### 1.1 In your Console

---

#### 1.1.1 Type and run the following code

```
my_building is a representation of the apartments on each floor of my 3 story building
my_building = [
 ['apt1a', 'apt1b', 'apt1c'],
 ['apt2a', 'apt2b', 'apt2c'],
 ['apt3a', 'apt3b', 'apt3c']
]
print("first floor: " + str(my_building[0]))
print("first floor, 3rd apartment: " + str(my_building[0][2])) //
```

### 1.2 In your Notebook

---

#### 1.2.1 Respond to the following

1. Write down what was printed.
2. How you would access the 2nd apartment of the 3rd floor (apt3b)?



# TEALS Program

[Home](#) | [Curriculum Map](#) | [Additional Readings](#) | [Change Log](#)

## 1 Lesson 4.05: Debugging and Quiz

---

### 1.1 Learning Objectives

---

Students will be able to...

- Read and understand longer programs involving loops
- Demonstrate their knowledge of looping, lists, and nested loops/lists
- Debug programs involving for loops and lists

### 1.2 Materials/Preparation

---

- Quiz (access protected resources by clicking on "Additional Curriculum Materials" on the [TEALS Dashboard](#))
- [Lab - Debugging Practice \(printable lab document\)](#) ([editable lab document](#))
- [Coding Example](#)
- Read through the quiz, lesson, and lab so that you are familiar with the requirements and can assist students
- Go through the quiz and create an answer key & scoring rubric

### 1.3 Pacing Guide

---

Duration	Description
25 Minutes	Quiz
10 Minutes	Lesson
20 Minutes	Lab
5 Minutes	Debrief

### 1.4 Instructor's Notes

---

#### 1.4.1 1. Quiz

- Have students start working on the Quiz.

#### **1.4.2 2. Lesson**

- Talk about how to read code in a loop (you can use something of your choice or the provided [Coding Example](#))
- Go over debugging practices for loops.
- Use print statements throughout the code to show where error is.
- Alter input to make sure body of loop is working correctly.
- Make sure lists don't go out of bounds/past the end of the list.

#### **1.4.3 3. Lab**

- This lab will have students start by reading code and looking for errors while **not** at the computer.
- After reading through the code they can practice the debugging practices mentioned on the computer to correct the programs.

#### **1.4.4 4. Debrief**

- Discuss as a class what was most helpful in debugging, highlight those practices throughout the upcoming project.

### **1.5 Accommodation/Differentiation**

---

Make sure to provide extended time on the quiz for any students that have that requirement in an IEP or 504 plan. Provide an [Additional Coding Example]) for students that finish the lab exercises early.

### **1.6 Forum discussion**

---

[Lesson 4.05: Debugging and Quiz \(TEALS Discourse Account Required\)](#)

formatted by [Markdeep 1.093](#) 



# TEALS Program

[Home](#) | [Curriculum Map](#) | [Additional Readings](#) | [Change Log](#)

## 1 Lesson 4.06: Tic Tac Toe

---

### 1.1 Learning Objectives

---

Students will be able to...

- Use project planning skills to complete a larger project
- Utilize loops, lists, and nested loops/lists to create a Tic-Tac-Toe game

### 1.2 Materials/Preparation

---

- Project Spec - Tic-Tac-Toe ([printable project Spec](#)) ([editable project spec](#))
- Alternate Project Spec - 2 Player Tic-Tac-Toe ([printable alternate project Spec](#)) ([editable alternate project spec](#))
- Tic Tac Toe Starter Code
- Solution (access protected resources by clicking on "Additional Curriculum Materials" on the [TEALS Dashboard])
- Read through the handout, lesson, and project so that you are familiar with the requirements and can assist students
- Try creating your own variation on the Tic-Tac-Toe game so you are familiar with the potential challenges and bugs your students will hit
- Review [4 Steps to Solve Any CS Problem](#)
- [Editable Grading Rubric](#)

### 1.3 Pacing Guide

---

#### 1.3.1 Day 1

Duration	Description
5 Minutes	Do Now
10 Minutes	Project Overview
30 Minutes	Planning/Design
10 Minutes	Debrief

#### 1.3.2 Days 2-9

Duration	Description
5 Minutes	Day Plan
10 Minutes	Review
35 Minutes	Project Work
5 Minutes	Debrief

## 1.4 Instructor's Note

---

### 1.4.1 1. Do Now

- Hand out the project spec and have students start reading through it.

### 1.4.2 2. Project Overview

- Demo the Tic-Tac-Toe game.
- Go over all of the game rules and program requirements.

### 1.4.3 3. Planning/Design

#### 1.4.3.1 Have students do the following in their notebook

- Write down the 4 Steps to Solve Any CS Problem from memory.
- Draw out game play and consider the overall design.
  - How will they represent the board?
  - How will they have users input their spots?
- Create function names for each user interaction.
- Figure out which variables are needed.

#### 1.4.3.2 Have students plan out their next 7 days (suggested plan below)

1. Set up the game board, basic game loop asking players for input, and dummy functions for each player's turn.
2. Create variables necessary to run the game, start implementing basic functions.
3. Focus on game play: 2 players should be able to play a game against each other.
4. Create functions for checking if the game is over. Create a horizontal checker, vertical checker.
5. Create the diagonal checkers.
6. Connect the functions together and test functionality.
7. Use multiple tests that game over check is correct, finalize project.

### 1.4.4 4. Debrief

- Take time to go over questions and confusion relating to project requirements.

- Make sure to look over individual student project plans and check that they have outlined the project in a way that makes sense.

#### **1.4.5 5. Day Plan**

- At the start of Days 2-9, have each student refer to their original project plan, write down what they hope to accomplish that day, and assess their schedule to see if they are on track.

#### **1.4.6 6. Review**

- Go over any parts of the program that the majority of the class is struggling with.
- Provide scaffolding and tips to students that are not on track for completion.

### **1.5 Accommodation/Differentiation**

---

- Students can create a variable sized board.
- The checkers can actually be done using one function, taking in the start x and y and the movement of the x and y.
- The planning phase of this project will be essential, especially for students who you think may struggle with this project.
- Provide more guidance and scaffolding to those students that need it.

### **1.6 Grading**

---

[Editable Grading Rubric](#)

#### **1.6.1 Objective Scoring Breakdown**

<b>Points</b>	<b>Percentage</b>	<b>Objective</b>	<b>Lesson</b>
6	19	Student uses for loops	4.01
3	10	Students can use the range and len functions	4.02
3	10	Students can use nested for loops	4.03
3	10	Student can use nested lists	4.04
5	16	Student can decompose a problem to create a program from a brief	
5	16	Student uses naming/ syntax conventions and comments to increase readability	
25		<b>Total Points</b>	

## 1.7 Forum discussion

---

Lesson 4.06: Project 4 (TEALS Discourse Account Required)

formatted by Markdeep 1.093 

# TEALS Program

[Home](#) | [Curriculum Map](#) | [Additional Readings](#) | [Change Log](#)

## 1 Project 4: Tic-Tac-Toe

---

Using Python, students will create a Tic-Tac-Toe game. This project has two parts

1. Designing the game so that two users can play Tic-Tac-Toe against one another.
2. Creating a Tic-Tac-Toe checker which will check the board to see if Xs or Os have won the game.

### 1.1 Overview

---

Tic-Tac-Toe is a game in which one player draws X's and another player draws O's inside a set of nine squares and each player tries to be the first to fill a row, column, or diagonal of squares with either X's or O's. We will be writing an interactive Tic-Tac-Toe program. At the end of each turn the computer will check to see if X's have won the game or if the O's have won the game.

#### 1.1.1 Behavior

- The program will prompt the user to enter their name and their opponents name.
- Whoever enters their name first will be playing as X's, and the other player will be O's.
- The players will take turns inputting the row and column they would like to place their mark.
- If that spot is already taken the program will ask for the spot again.
- At the end of each player's turn the program will
  - check if that player has won.
  - print the updated game board.
- If there are no more spots open and nobody has won the game, the program will print Tie game!.

#### 1.1.2 Implementation Details

- Use variables to store the user names for personalized prompts.
- Create a game board represented as a list of lists, size 3 by 3.

**Note: This is a change from our earlier implementations of Tic-Tac-Toe. Why do you think this might be better?** \* Check for a winner horizontally, vertically, and on both diagonals. \* Cannot allow a user to overwrite a spot on the board.





# TEALS Program

[Home](#) | [Curriculum Map](#) | [Additional Readings](#) | [Change Log](#)

## 1 Unit 5 - Music Programming (Optional Unit)

---

This unit is a tremendous unit to learn how to use Python to code Music. If this is a stand alone computer science course, it is a great way to expose students to a great example of the power of Python. If you are preparing your students for the AP Computer Science A course and/or a certification, we recommend prioritizing units 6 and 7 and coming back to unit 5 if there is time at the end of the semester.

### 1.1 Essential Questions

---

- What is Modulo and how can you apply it to a loop in Python?
- How do you loop through the items in a list?

### 1.2 Pacing Guide

---

#### 1.2.1 Timing

1 Day = 50 minute class period

Lesson	Days	Resources
5.01:	1	<a href="#">5.01 Slide Deck</a>
5.02:	1	<a href="#">5.02 Slide Deck</a>
5.03:	1	<a href="#">5.03 Slide Deck</a>
5.04:	1	<a href="#">5.04 Slide Deck</a>
5.05:	5	<a href="#">5.05 Slide Deck</a>
<b>Total Days</b>	9	
<b>Total Minutes</b>	450	

### 1.3 Key Terms

---

- Digital Audio Workstation
- Sound tab
- fitMedia()

- setTempo()
- Modulo
- Section
- A-B-A form

*formatted by Markdeep 1.093 ↗*

# TEALS Program

[Home](#) | [Curriculum Map](#) | [Additional Readings](#) | [Change Log](#)

## 1 Lesson 5.01: Earsketch Intro

---

**Note:** Unit 5 involves a tool called EarSketch from Georgia Tech. EarSketch is a Digital Audio Workstation with an embedded scripting environment that supports Python or JavaScript. Using EarSketch, students can create songs by writing Python code. All of the Earsketch activities require you to use the EarSketch Editor instead of the IDE you have been using so far (e.g. Repl.it).

### 1.1 Learning Objectives

---

Students will be able to...

- Define and identify: **Digital Audio Workstation (DAW)**, **sound tab**, `fitMedia()`, `setTempo()`
- Demonstrate beats using the above functions
- Demonstrate Looping through items in a list

### 1.2 Materials/Preparation

---

- Do Now
- Lab - Intro to EarSketch ([printable lab document](#)) ([editable lab document](#))
- EarSketch Editor
- Associated Reading in EarSketch
- Read through the do now, lesson, and lab so that you are familiar with the requirements and can assist students

### 1.3 Pacing Guide

---

Duration	Description
5 Minutes	Do Now
15 Minutes	Lesson
30 Minutes	Lab
5 Minutes	Debrief

### 1.4 Instructor's Notes

---

## 1.4.1 1. Do Now

- Display the Do Now on the board.
- Students are introduced to EarSketch and the functions `setTempo()` and `fitMedia()`.

## 1.4.2 2. Lesson

### 1.4.2.1 Activity

Copy the following code from the reading above into the [EarSketch Editor]:

```
script_name: Intro_Script
#
author: The EarSketch Team
#
description: This code adds one audio clip to the DAW
#
#
#
#Setup Section
from earsketch import *
init()
setTempo(120)

#Music Section
fitMedia(TECHNO_SYNTHPLUCK_001, 1, 1, 9)

#Finish Section
finish() //
```

- Have students write in their notebooks an answer to the following
- What inputs does `fitMedia` take? Press the run button and describe what happened.

## 1.4.3 In their Notebooks, ensure students have the following terms defined

### 1.4.3.1 Digital Audio Workstation (DAW)

- a piece of computer software for recording, editing, and playing digital audio files.
- Audio files store information that the computer uses to play back music.
- In the context of a DAW, these audio files are called clips.
- The DAW allows you to edit and combine multiple clips on a musical timeline
- Two popular DAWs used in the music industry are Pro Tools and Logic Pro.

### 1.4.3.2 Sound Browser

**Sounds Tab:** Here, you can browse and search a collection of short pre-made audio clips for you to use in your music.

**Scripts Tab:** A list of your saved EarSketch projects. Each script is a separate music project. Click a project title to open it in a new tab (in the code editor panel).

**API Tab:** A description of every EarSketch function. We will return to this in later lessons.

**Code Editor:** A text editor with numbered lines. Type your code here, press “Run”, and it will turn into music in the DAW.

### **1.4.3.3 Console:**

`setTempo()`: comes with a default argument of 120, but let's change it to 100. This sets the tempo of our project to 100 beats per minute. As you can see, the name of a function tells you what it does.

`fitMedia()`: to add sound to the DAW. `fitMedia()` requires four arguments: clip name; track number; starting measure; ending measure. In other words, you tell it the name of the audio clip you want to add to the DAW, which track to put it on, and which measures to put it between. For now, just type in `fitMedia()` without any arguments.

### **1.4.4 3. Lab**

- Practice updating songs by entering different arguments into the `fitMedia()` function.

### **1.4.5 4. Debrief**

- Talk about any of the phrases or issues the students had.
- Take time to review the key terms and functions introduced today
- Have students read the next part(s) of the guide in EarSketch documentation.

## **1.5 Accommodation/Differentiation**

---

Students that are moving quickly can read additional documentation on the EarSketch website in order to move ahead and expand their skills and understanding.

Some students may need reminders about certain music terminology (tempo, measure, beat, etc.).

## **1.6 Forum discussion**

---

[Lesson 5.01: Earsketch Intro \(TEALS Discourse Account Required\)](#)

formatted by [Markdeep\\_1.093](#) 

# TEALS Program

[Home](#) | [Curriculum Map](#) | [Additional Readings](#) | [Change Log](#)

## 1 Do Now 5.01

---

### 1.1 Introduction to EarSketch

---

EarSketch, created by Georgia Tech, is an online tool that allows you to learn Python and music technology side by side. By applying concepts already covered in this curriculum such as functions, loops, and conditionals, you will be able to use the EarSketch IDE to create unique and entertaining beats and songs!

1. Read through and participate in the following lessons in [EarSketch Editor]
  - o 1.1.2 [Tools of the Trade: DAWs and APIs](#)
  - o 1.1.3 [The EarSketch Workspace](#)
  - o 1.1.5 [The DAW in Details](#)
  - o 1.1.8 [Creating a New Script](#)
  - o 1.1.9 [Composing in EarSketch - fitMedia\(\)](#)
2. In your notebook, write down any terms or phrases that stand out to you to remember as you work through the lessons

### 1.2 Location of Lessons

---

*Image of Earsketch Location*

formatted by [Markdeep 1.093](#) 

# TEALS Program

[Home](#) | [Curriculum Map](#) | [Additional Readings](#) | [Change Log](#)

## 1 Lab 5.01 - Intro to EarSketch

---

Create a basic [EarSketch] project using the following requirements:

- Open the Earsketch DAW and log in.
- Click “Options – Save Script to Cloud” and set a name for your song.
- In comments at the top of your code, include your name, the name of the song (e.g. “Lab 5.01”), and a brief, clear description of the project.
- Set the tempo of the project to 140 BPM.
- Make 4 different variables that will hold the following constants:
  - DUBSTEP\_BASS\_WOBBLE\_007
  - DUBSTEP\_PAD\_003
  - DUBSTEP\_DRUMLOOP\_MAIN\_005
  - DUBSTEP\_LEAD\_004
- Make at least 4 tracks using the fitMedia() function. Each track should contain a different audio loop from the other tracks.
- Each of the tracks should start or stop on different measures (e.g. track 1 starts on measure 1, track 2 starts on measure 4, and so on).
- Define and use at least 4 variables using 4 different constants from the EarSketch Audio Loop Browser.
- The project should have a total length of 12 measures.
- Include descriptive comments throughout your script.
- Copy and save the script to your online IDE for turn in.

### 1.1 Copyright Note

---

The above is taken from the EarSketch teaching resources.

# TEALS Program

[Home](#) | [Curriculum Map](#) | [Additional Readings](#) | [Change Log](#)

## 1 Lesson 5.02: EarSketch Music

---

### 1.1 Learning Objectives

---

Students will be able to...

- Define and identify: **rhythm, beat, tempo, measures**, `setEffect()`, `makeBeat()`
- Demonstrate beats using the functions
- Demonstrate a loop through items in a list

### 1.2 Materials/Preparation

---

- [Do Now](#)
- [Lab - EarSketch Music \(printable lab document\)](#) ([editable lab document](#))
- [EarSketch Editor](#)
- Associated Reading in EarSketch
- Read through the do now, lesson, and lab so that you are familiar with the requirements and can assist students

### 1.3 Pacing Guide

---

Duration	Description
5 Minutes	Do Now
10 Minutes	Lesson
35 Minutes	Lab
5 Minutes	Debrief

### 1.4 Instructor's Notes

---

#### 1.4.1 1. Do Now

- Students should be given time to read unit 2 of the EarSketch documentation.
- Students should answer the questions included in the do now and be prepared to discuss them as a class.

## 1.4.2 2. Lesson

- Call on students to discuss the answers to the questions from the Do Now.

### 1.4.2.1 Recap the following key concepts from the reading

- **Rhythm:** describing how the music moves through time.
- **beat** is the basic unit of time in music.
  - clapped along to a song, you are clapping on each beat.
  - The length of a beat depends on the overall speed of the song, called the *tempo*.
  - Beats are grouped into **measures**. In EarSketch, measures always have four beats.
- `makeBeat()`: instead of composing at the measure-level, we can work at the note-level.
- **parameters**: clip name, track number, measure number, beat string
- **Tempo** is measured in beats per minute (bpm).
  - Clapping at 60 bpm, each beat lasts one second.
  - At 120 bpm, each beat takes half a second.
  - The higher the bpm, the faster the song and the shorter the duration of each beat.
- `setEffect()`: add an effect to a track.
  - **Takes parameters:** track number, effect name, effect parameter, effect value

## 1.4.3 3. Lab

- Follow the EarSketch instructions in the lab to use the `makeBeat()` function
- Create a simple song with 2 uses of `fitMedia()`, 2 uses of `makeBeat()` and 1 use of an effect.

## 1.4.4 4. Debrief

- Talk about the new functions learned today, and go over any questions about data types and using strings.
- Have students write down two things they have learned so far in Earsketch.

## 1.5 Accommodation/Differentiation

---

Students can use looping and if statements to their song as an extension activity to make their songs more complex.

Students will likely bring a wide range of background knowledge around music and the related terminology. Offer additional support to those students that are less familiar with the terms being introduced in this lesson.

## 1.6 Forum discussion

---

Lesson 5.02: EarSketch Music (TEALS Discourse Account Required)

# TEALS Program

[Home](#) | [Curriculum Map](#) | [Additional Readings](#) | [Change Log](#)

## 1 Do Now 5.02

---

1. Read through and participate in the following lessons in [EarSketch Editor]
  - 1.2.0 [Rhythm](#)
  - 1.4.0 [Using Effect in EarSketch - setEffect\(\)](#)
  - 1.7.1 [Copyright](#)
  - 2.11 [Making Custom Beats: makeBeat](#)
2. In your notebook, write down answers the following questions:
  - What are three examples of effects used in music production and available in EarSketch?
  - Write down how you would use the `setEffect()` function to change the volume level of track 4 to a value of `-15.0`.
  - For most songs, how many beats make up one measure?
  - What is the purpose of the `0`, `+`, and `-` symbols when used in an EarSketch beat strings

formatted by [Markdeep 1.093](#) 

# TEALS Program

[Home](#) | [Curriculum Map](#) | [Additional Readings](#) | [Change Log](#)

## 1 Lab 5.02: EarSketch Music

---

### 1.1 Beat Strings

---

#### 1.1.1 In your Notebook

Beat strings are made up of combinations of three different characters. In your Notebook, Explain the rhythmic meaning of each of the characters.

1. "0"
2. "+"
3. "-"

#### 1.1.2 Lab Activity

Create a new EarSketch project using the following requirements

1. Include your name and a project description at the top of the file.
2. This project should have 4 tracks and be 12 measures in length.
3. Each track should use a different/unique audio sample.
4. Create 4 variables that hold audio loop constants.
5. Create 1 variable that holds a beat string.
6. On one track, use the makeBeat() function with your beat string.
  - Your beat string rhythm should repeat for all 12 measures of the project.
7. Use the setEffect() function for at least one track.
8. Make sure to use comments that outline and describe your code.

#### 1.1.3 Copyright Note

The above is taken from the Earsketch teaching resources.

# TEALS Program

[Home](#) | [Curriculum Map](#) | [Additional Readings](#) | [Change Log](#)

## 1 Lesson 5.03: Earsketch Control Flow

---

### 1.1 Learning Objectives

---

Students will be able to...

- Define and identify: **modulo**
- Demonstrate looping and control structures
- Demonstrate the use of looping concepts in music making via EarSketch
- Demonstrate using control structures to create music

### 1.2 Materials/Preparation

---

- Do Now
- Lab - EarSketch Control Flow ([printable lab document](#)) ([editable lab document](#))
- EarSketch Editor
- Associated Reading in EarSketch
- Read through the do now, lesson, and lab so that you are familiar with the requirements and can assist students
- Video Explanation of Modulo in Python (Visual Rhythms)

[

](<https://youtu.be/2Tg9FxIajho>)

*Video Explanation of Modulo in Python*

### 1.3 Pacing Guide

---

Duration	Description
5 Minutes	Do Now
10 Minutes	Lesson
35 Minutes	Lab
5 Minutes	Debrief

### 1.4 Instructor's Notes

---

#### **1.4.1 1. Do Now**

- Students should be given time to read unit 3 of the EarSketch documentation.
- Students should answer the questions included in the do now and be prepared to discuss them as a class.

#### **1.4.2 2. Lesson**

- Call on students to discuss the answers to the questions from the Do Now.
- Review looping in Python. Look at the examples given in the EarSketch documentation and play them for the class.
- Review if statements and control flow. As with looping go over examples (starting at section "Conditional Statements in Loops"). Focus on the **modulo** operator, reminding students that it is an operator that returns the remainder after division.

#### **1.4.3 3. Lab**

- Students practice using looping to make a song a single track. Make sure that students use if statements and the modulo operator.
- Ask the students to practice using looping, effects, and control flow structures while using 2-3 different tracks.

#### **1.4.4 4. Debrief**

- Talk about any of the common issues that students had with loops and control flow.

### **1.5 Accommodation/Differentiation**

---

Students that are moving quickly can read additional documentation on the EarSketch website in order to move ahead and expand their skills and understanding.

### **1.6 Forum discussion**

---

[Lesson 5.03: Earsketch Control Flow \(TEALS Discourse Account Required\)](#)

formatted by [Markdeep 1.093](#) 

# TEALS Program

[Home](#) | [Curriculum Map](#) | [Additional Readings](#) | [Change Log](#)

## 1 Do Now 5.03

---

1. Read through and participate in the following lessons in [EarSketch Editor]
  - o 2.4.0 [for loop](#)
  - o 3.1.2 [Conditional Statements](#)
  - o 3.3.0 [Procedure - modulo](#)
2. In your notebook, answer the following questions, based on the reading:
  - o Describe how using a `for` loop can be useful in using EarSketch to create a new beat.
  - o How can an `if` statement be used with the **modulo** operator in order to have a beat repeat *every other measure*?

formatted by [Markdeep 1.093](#) 

# TEALS Program

[Home](#) | [Curriculum Map](#) | [Additional Readings](#) | [Change Log](#)

## 1 Lab 5.03: Earsketch Control Flow

---

1. Write a for loop that repeats the given rhythm every other measure from measures 1 to 8 (e.g. 1, 3, 5, etc.) on track 1.
  - o Audio loop constant: HIPHOP\_STOMP\_BEAT\_001
  - o Rhythm: "0—0—0—0000+++"
2. Create an EarSketch script using the following requirements. Include your name and a description in comments at the top of the file.
  - o Create a project with 4 tracks that is at least 16 measures in length.
  - o 2 tracks should be music tracks, containing `fitmedia()` function calls.
  - o The remaining two tracks should contain `makeBeat()` function calls.
  - o One track should use a for loop to repeat a rhythm (beat string) for every measure in the song.
  - o The other track should use a for loop to repeat a rhythm for every other measure in the song (e.g. 1, 3, 5, 7, 9, 11...).

### 1.1 Copyright Note

---

The above is taken from the Earsketch teaching resources.

formatted by [Markdeep 1.093](#) 

# TEALS Program

[Home](#) | [Curriculum Map](#) | [Additional Readings](#) | [Change Log](#)

## 1 Lesson 5.04: EarSketch User-Defined Functions

---

### 1.1 Learning Objectives

---

Students will be able to...

- Define and identify: **abstraction, section, A-B-A form**
- Create and apply user-defined functions to create songs with complicated form

### 1.2 Materials/Preparation

---

- [Do Now](#)
- [Lab - User-Defined Functions \(printable lab document\)](#) ([editable lab document](#))
- [EarSketch Editor](#)
- Associated Reading in EarSketch
- Read through the do now, lesson, and lab so that you are familiar with the requirements and can assist students

### 1.3 Pacing Guide

---

Duration	Description
5 Minutes	Do Now
10 Minutes	Lesson
35 Minutes	Lab
5 Minutes	Debrief

### 1.4 Instructor's Notes

---

#### 1.4.1 1. Do Now

- Students should be given time to read unit 4 of the EarSketch documentation.
- Students should answer the questions included in the do now and be prepared to discuss them as a class.

## 1.4.2 2. Lesson

- Call on students to discuss the answers to the questions from the Do Now.
- Recap the following key concepts from the reading:
  - **Section:** generally refers to several measures of music (often 2, 4, 8, or 16 measures) that sound like a single musical unit.
  - **A-B-A Form:** a song divided into three sections:
    - Section A: measures 1-4.
    - Section B: measures 5-7. Features contrasting sounds to Section A.
    - Section A (repeated): measures 7-10.
- Ask students about returning the measure number.
- Go over the concept of passing around the measure number and make sure students understand how it would be helpful.
- Since it allows you to change how long/short sections are without updating every function

## 1.4.3 3. Lab

- Demo a song with **A-B-A Form**:

[

](<https://www.youtube.com/watch?v=PSZxmZmBfnU>)

*Somewhere over the Rainbow*

- Students create their own song with the form ABABAB.

## 1.4.4 4. Debrief

- Talk about any of the phrases or issues the students had.
- If time allows, give students the opportunity to demo their songs for the class.

## 1.5 Accommodation/Differentiation

---

Students that are moving quickly could explore and create alternative song forms (ex. AABA).

## 1.6 Forum discussion

---

[Lesson 5.04: EarSketch User-Defined Functions \(TEALS Discourse Account Required\)](#)

formatted by [Markdeep 1.093](#) 

# TEALS Program

[Home](#) | [Curriculum Map](#) | [Additional Readings](#) | [Change Log](#)

## 1 Do Now 5.04

---

1. Begin reading the following lessons of the EarSketch documentation in [Unit 2](#).
  - o 2.1.0 [Sections and Form](#)
  - o 2.1.1 [A-B-A form](#)
  - o 2.1.2 [Custom Functions](#)
2. Answer the following questions, based on the reading:
  - o Why might it be useful to group sections of music into functions?
  - o Write the code for defining a `verse()` function that takes in 5 parameters: `drums`, `guitar`, `bass`, `startMeasure`, and `endMeasure`.

*formatted by [Markdeep 1.093](#) ↗*

# TEALS Program

[Home](#) | [Curriculum Map](#) | [Additional Readings](#) | [Change Log](#)

## 1 Lab 5.04 - User-Defined Functions

---

Create a new EarSketch project that meets the requirements below. Include your name and project description at the top of the file.

- Create two functions. Both functions should use `fitMedia()` and `makeBeat()` at least once.
  - `sectionA()` function
    - Use parameters for the audio clip variable(s), starting measures, and ending measures.
    - This function should have at least 3 tracks.
  - `sectionB()` function
    - Use parameters for starting and ending measures.
    - This function should have at least 2 tracks.
- Call both functions in an alternating pattern at least three times (ABABAB)
- Use comments to outline and describe your code.

### 1.1 Copyright Note

---

The above is taken from the Earsketch teaching resources.

formatted by [Markdeep 1.093](#) 

# TEALS Program

[Home](#) | [Curriculum Map](#) | [Additional Readings](#) | [Change Log](#)

## 1 Lesson 5.05: EarSketch Project

---

### 1.1 Learning Objectives

---

Students will be able to...

- Create a complete song in EarSketch with multiple parts
- Utilize EarSketch's features and functions

### 1.2 Materials/Preparation

---

- [Do Now](#)
- Solution (access protected resources by clicking on "Additional Curriculum Materials" on the [TEALS Dashboard](#))
- [EarSketch Editor](#)
- [Project \(printable project Spec\)](#) ([editable project spec](#))
- Read through the do now and project spec so that you are familiar with the requirements and can assist students
- Practice creating your own EarSketch song(s) to demonstrate to students and to better understand the challenges they may face in the project
- Review [4 Steps to Solve Any CS Problem](#)

### 1.3 Day 1 Pacing

---

Duration	Description
5 Minutes	Do Now
10 Minutes	Project Overview
15 Minutes	Project Planning
25 Minutes	Begin Project

### 1.4 Days 2-5 Pacing

---

Duration	Description
5 Minutes	Do Now
10 Minutes	Topic Review
35 Minutes	Project Work
5 Minutes	Debrief

## 1.5 Instructor's Notes

---

### 1.5.1 1. Do Now

- Display the Do Now on the board
- For Days 2-5, the Do Now is time for students to write down issues they had with the project from the day before and what they plan on doing to fix those issues.
- Students should take time to create a timeline for when certain tasks will be completed.

### 1.5.2 2. Project Overview

- Review the terminology, topics, and skills that students have learned from this unit. Talk about any questions or things the students are struggling with.
- Discuss the parts of the song mentioned in the Do Now (chorus, bridge, and verses) and how they fit into building a song.
- Distribute the project spec and talk students through the requirements and scoring rubric.
- Demo a final song for the students to see a finished product.

### 1.5.3 3. Project Planning

- Instruct students to create a project plan for what specifically they will accomplish during each day of the project.
- Take time to check that each student has created a project plan before they begin working on their song.

## 1.6 Accommodation/Differentiation

---

Certain students that have a limited music background may need additional assistance during the planning phase of the project. Students may need additional examples demonstrating the difference between a verse, chorus, and bridge.

## 1.7 Grading

---

### 1.7.1 Scheme/Rubric

<b>Functional Correctness(Behavior)</b>	
Song Runs and Plays	5
Recognizable Chorus vs Verse	10
Correct Length	5
Contains some reoccurring themes	5
<b>Sub total</b>	25
<b>Technical Correctness</b>	
Correct use of loop	5
Correctly uses control flow	5
Correctly use of <code>fitMedia</code> , <code>makeBeat</code> , <code>setEffect</code>	10
Use of user defined functions for choruses, forms, verses	10
Keeps track of measure using return statements	15
<b>Sub total</b>	45
<b>Total</b>	70

## 1.8 Forum discussion

---

Lesson 5.05: EarSketch Project (TEALS Discourse Account Required)

formatted by Markdeep 1.093 ↗

# TEALS Program

[Home](#) | [Curriculum Map](#) | [Additional Readings](#) | [Change Log](#)

## 1 Do Now 5.05

---

1. Begin reading the following lessons in the EarSketch documentation in Unit 3.
  - o 3.5.3 [Sharing an EarSketch Script](#)
2. Write down 2-3 songs that have a structure that you could emulate in EarSketch.
3. For one of those songs write down the different musical parts of the song, in order. This most likely will include verses, the chorus, and the bridge.

### 1.1 Example

---

Using Pharrell Williams' "Happy":

- Section A
- Section B
- Section A
- Section B
- Bridge
- Section B
- Section B
- Bridge
- Section B
- Section B

Note: No intro or outro in this song.

### 1.2 Copyright Note

---

The above is taken from the Earsketch teaching resources.

# TEALS Program

[Home](#) | [Curriculum Map](#) | [Additional Readings](#) | [Change Log](#)

## 1 Project 5: EarSketch Song

---

### 1.1 Overview

---

Using Python and EarSketch you will create a complete song consisting of 2 verses, a chorus, and a bridge. We have worked in class to practice creating sections of songs. We'll use that knowledge to create a song using the samples we are given.

#### 1.1.1 Behavior

- The song will have pattern (verse, chorus, verse2, chorus, bridge, chorus)
- Create 4 different variables that will hold your choice of audio constants.
- The verses should contain some of the same forms, but be independent.
- The project should have at least 4 tracks, and each track should use a different/unique sound file (different constant).
- Each verse should consist of at least 8 measures.
- The chorus should consist of at least 8 measures.
- The bridge should consist of 2-8 measures.
- The song should have a flow and underlying themes that recur via the use of variables.

#### 1.1.2 Implementation Details

- Use of `for` loop for creating tracks
- Use of control flow operators
- Proper use of `fitMedia`, `makeBeat`, `setEffect`
- Use of user-defined functions for tracking forms and verses and choruses
- Use of return statements for tracking the measure

### 1.2 Copyright Note

---

The above is adapted from the Earsketch teaching resources.

# TEALS Program

[Home](#) | [Curriculum Map](#) | [Additional Readings](#) | [Change Log](#)

## 1 Unit 6 - Dictionaries

---

### 1.1 Essential Questions

---

- How do you use Dictionaries to create key-value pairs in Python?
- How do you add, remove and append list values in a dictionary?
- How do you use loops to traverse the key/value pairs of a dictionary?

### 1.2 Pacing Guide

---

#### 1.2.1 Timing

1 Day = 50 minute class period

Lesson	Days	Resources
6.01:	1	<a href="#">6.01 Slide Deck</a>
6.02:	1	<a href="#">6.02 Slide Deck</a>
6.03:	1	<a href="#">6.03 Slide Deck</a>
6.04:	1	<a href="#">6.04 Slide Deck</a>
6.05:	7	<a href="#">6.05 Slide Deck</a>
<b>Total Days</b>	11	
<b>Total Minutes</b>	550	

### 1.3 Key Terms

---

- Dictionary
- Key
- Value
- Pop
- Default Value
- Update
- Append

- Remove

*formatted by Markdeep 1.093 ↗*

# TEALS Program

[Home](#) | [Curriculum Map](#) | [Additional Readings](#) | [Change Log](#)

## 1 Lesson 6.01: Introduction to Dictionaries

---

### 1.1 Learning Objectives

---

Students will be able to...

- Define and identify: **dictionary, key, value**
- Create dictionaries of key-value pairs
- Access items from dictionaries

### 1.2 Materials/Preparation

---

- [Do Now](#)
- [Lab - Dictionary of Internet Abbreviations](#) ([printable lab document](#)) ([editable lab document](#))
- [Associated Reading](#)
- Read through the do now, lesson, and lab so that you are familiar with the requirements and can assist students

### 1.3 Pacing Guide

---

Duration	Description
5 Minutes	Do Now
10 Minutes	Lesson
35 Minutes	Lab
5 Minutes	Debrief

### 1.4 Instructor's Notes

---

#### 1.4.1 1. Do Now

- Display the Do Now on the board.
- Students will copy and edit code involving creating a dictionary and accessing items from that dictionary.

## 1.4.2 2. Lesson

### 1.4.2.1 Instruction - Dictionaries

- Ask the students what type they think `my_dictionary` is.
- `my_dictionary` is a **dictionary** or a collection of **key-value** pairs.
- You use the key to look up the value in the dictionary.
- Keys and values can be of any type. The syntax is: `{key : value, key : value, ...}`

### 1.4.2.2 Discussion

- Did anyone run the `type()` function to find out what type 'my\_dictionary' is?
- Ask: what are the keys in the example from the Do Now? What are the associated values?
- Ask the students what `my_dictionary['dog']` did, and if this syntax reminds them of anything (lists!).

### 1.4.2.3 Instruction - Square Brackets

- To get the value associated with a key in a dictionary you use square brackets.
- You can also use `my_dictionary.get()`, which will return `None` if the key isn't there.
- Note:* You can pass in a second argument to `get` which takes the place of the `None` default.

### 1.4.2.4 More Discussion

- Ask how students would get the value for `chair` or `car`.
- Discuss what happened when students ran `my_dictionary['kittens']`?

### 1.4.2.5 'in' keyword

- Explain that this error is common and means that there is no value in the dictionary. To avoid this error, use the `in` keyword with an `if` statement. If a certain key is in a specified dictionary, it will return `true`. Otherwise it will return `false`.

### 1.4.2.6 Example

```
my_dictionary = {'a': 1, 'b': 2, 'c': 3}
if 'a' in my_dictionary:
 print("It's there!")
else:
 print("It's missing!")
```



## 1.4.3 3. Lab

- Students will create a dictionary translating common internet phrases into their meanings.

## 1.4.4 4. Debrief

- Review what was covered in today's lesson and check for understanding of the three concepts covered: **dictionaries**, **keys**, and **values**.

## **1.5 Accommodation/Differentiation**

---

- If any students are struggling with today's lesson, be prepared to offer additional examples of the usefulness of having key-value pairs.
- Students that are moving quickly through the lab should work on the bonus and research how to add new key/value pairs to a dictionary.

## **1.6 Forum discussion**

---

Lesson 6.01: Introduction to Dictionaries (TEALS Discourse Account Required)

*formatted by Markdeep 1.093* 



# TEALS Program

[Home](#) | [Curriculum Map](#) | [Additional Readings](#) | [Change Log](#)

## 1 Lab 6.01 - Internet Abbreviations

---

### 1.1 Instructions

---

Write a program that uses a dictionary to offer users the meanings of common internet abbreviations.

The program, `dictionary_lab.py`, prompts the user to enter an internet abbreviation they would like explained. If the requested abbreviation is in the program's dictionary (use the `in` keyword with an `if` statement to test this), then it prints out the definition. If the abbreviation is not in the dictionary, the program prints an apologetic message saying that it could not find a definition.

#### 1.1.1 Example Output

```
>>> python3 dictionary_lab.py

What word would you like to look up? nbd
nbd: a phrase meaning no big deal

What word would you like to look up? kittens
Sorry, kittens is not defined

What would would you like to look up?
```



### 1.2 Bonus

---

Extend the program with any of these features:

The user can

- update the definitions (values) for existing abbreviations in the dictionary
- add new abbreviations (keys) and provide their definitions (values).
- delete entries (key, value pairs) from the dictionary.
- get the entire dictionary printed to the screen.

Lesson 6.01 did not cover all the techniques for manipulating dictionaries that you will need to program these features. Search for the necessary information in the? and the?.

# TEALS Program

[Home](#) | [Curriculum Map](#) | [Additional Readings](#) | [Change Log](#)

## 1 Lesson 6.02: Dictionaries Methods

---

### 1.1 Learning Objectives

---

Students will be able to...

- Define and identify: **pop, default value**
- Update values in a dictionary
- Add values to a dictionary
- Remove values from a dictionary

### 1.2 Materials/Preparation

---

- [Do Now](#)
- [Lab - Word Counter \(printable lab document\)](#) ([editable lab document](#))
- Solution (access protected resources by clicking on "Additional Curriculum Materials" on the [TEALS Dashboard](#))
- [Associated Reading](#)
- Read through the do now, lesson, and lab so that you are familiar with the requirements and can assist students

### 1.3 Pacing Guide

---

Duration	Description
5 Minutes	Do Now
10 Minutes	Lesson
35 Minutes	Lab
5 Minutes	Debrief

### 1.4 Instructor's Notes

---

#### 1.4.1 1. Do Now

- Display the Do Now on the board.

- Students type code that edits dictionaries by modifying, adding, or removing key:value pairs.

## 1.4.2 2. Lesson

### 1.4.2.1 Part 1 of the Do Now

- Discuss what the students learned from and go over what the code is doing.
- Describe that you can update a value in a dictionary in a similar way to how you would update a value in a list.

### 1.4.2.2 Part 2 of the Do Now

- Discuss what the students learned from and go over what the code is doing.
- Note that you can add values to a dictionary by using the same syntax.
- Review: How might you check if a value was in a dictionary before adding it?

### 1.4.2.3 Part 3 of the Do Now

- Discuss what the students learned from and go over what the code is doing.
- Review what **pop** does, as well as what the second argument does.
- This is the value that will be returned if the first argument is not in the dictionary.

## 1.4.3 3. Lab

- Students will create a word count algorithm that will count the number of words in a list of words.

## 1.4.4 4. Debrief

- Talk about any confusion the students had. Discuss additional ideas of how dictionaries might be useful.

## 1.5 Accommodation/Differentiation

---

Some students may need to be explicitly reminded of the `in` keyword to check if a certain key is in their dictionary.

## 1.6 Forum discussion

---

[Lesson 6.02: Dictionaries Methods \(TEALS Discourse Account Required\)](#)

# TEALS Program

[Home](#) | [Curriculum Map](#) | [Additional Readings](#) | [Change Log](#)

## 1 Do Now 6.02

---

### 1.1 1. In your Console

---

#### 1.1.1 Type the following

```
my_dictionary = {
 'kittens': 'cute animals'
}
my_dictionary['kittens'] = 'p. cute'
print(my_dictionary)
```



### 1.1.2 In your Notebook

#### 1.1.2.1 Respond to the following

Write down what the 2nd line does.

### 1.2 2. In your Console

---

```
my_dictionary = {}
my_dictionary['puppies'] = 'baby dogs'
print(my_dictionary)
```



### 1.2.1 Continue in your Notebook

#### 1.2.1.1 Respond to the following prompt

Write down what the 2nd line does.

### 1.3 3. In your Console

---

```
my_dictionary = {
 'kittens': 'cute animals',
 'puppies': 'baby dogs'
}
```

```
my_dictionary.pop('kittens')
print(my_dictionary)
my_dictionary.pop('bunnies')
my_dictionary.pop('bunnies', None)
```



### 1.3.1 Continue In your Notebook

1. Write down what the second line does.
2. What is different between `my_dictionary.pop('bunnies')` and `my_dictionary.pop('bunnies', None)`?

formatted by [Markdeep 1.093](#) 

# TEALS Program

[Home](#) | [Curriculum Map](#) | [Additional Readings](#) | [Change Log](#)

## 1 Lab 6.02 - Word Counter

---

In this lab we will implement a word frequency algorithm. It will tell you how many of each word you had in an essay.

At the top of the document save a variable with a long paragraph (example below). In order to turn this paragraph into a list of lower case words we will use the `split(" ")`, `replace()`, and `lower()` functions. There is code at the bottom of this page that will do this for you. Feel free to read more about `split()` in the Python documentation, but it's not critical to this lab.

For each word in the document, count the number of times it occurs. Consider the following phrase: 'Cats are cool. Baby cats are called kittens. Cats make great pets.' The word 'cats' appears 3 times. The word 'are' appears 2 times.

The program will first create a dictionary with the words as keys and the number of times they occur as values. Then it will prompt the user which word they are curious about. If the word was in the paragraph it will print the number of times it occurred.

### 1.1 Example

---

```
>>> python3 word_frequency_lab.py
What word would you like to know the frequency of? cats
'cats' occurs 3 times
```

```
>>> python3 word_frequency_lab.py
What word would you like to know the frequency of? dogs
'dogs' does not occur
```

#### 1.1.1 split, replace, and lower

This is the code to lower case the letters in the paragraph, remove the periods, and split them into individual words.

```
example_paragraph = "It was a beautiful day in New York City. Our hero Ariana Grande was on a walk from the Standard to Duane Reade. Ariana Grande was walking rather quickly because she had lived in New York for a few months. All of a sudden a slimy donut appeared out of nowhere. Ariana Grande decided to prance foolishly instead of dealing with the situation. Thrown off from Duane Reade Ariana Grande decides to go to Times Square instead. What a beautiful day in New York."
```

```
#make all letters lowercase
example_paragraph_lower = example_paragraph.lower()

#remove all periods
example_paragraph_lower_no_punctuation = example_paragraph_lower.replace(".", "")

#convert paragraph into a list of individual strings
example_word_list = example_paragraph_lower_no_punctuation.split(" ")
```



# TEALS Program

[Home](#) | [Curriculum Map](#) | [Additional Readings](#) | [Change Log](#)

## 1 Lesson 6.03: Dictionaries of Lists

---

### 1.1 Learning Objectives

---

Students will be able to...

- Create dictionaries with keys and values of different types
- Update, append, or remove list values in a dictionary

### 1.2 Materials/Preparation

---

- [Do Now](#)
- [Lab - Dictionaries Storing Lists \(printable lab document\)](#) ([editable lab document](#))
- [Associated Reading](#)
- Read through the do now, lesson, and lab so that you are familiar with the requirements and can assist students

### 1.3 Pacing Guide

---

Duration	Description
5 Minutes	Do Now
10 Minutes	Lesson
35 Minutes	Lab
5 Minutes	Debrief

### 1.4 Instructor's Notes

---

#### 1.4.1 1. Do Now

- Display the Do Now on the board.
- Students will explore how to use dictionaries containing lists, as well as how to add values to those lists.

## 1.4.2 2. Lesson

### 1.4.2.1 Discuss part 1 of the Do Now

- Go over the type of `weekend_dates`.
- It is still a dictionary type. In this case the dictionary goes from string (key) to list (value).
- Dictionaries can also go from numbers to lists, or numbers to strings, or any other combination.

### 1.4.2.2 Discuss part 2 of the Do Now

- Review how you update a value within a dictionary, as well as how to append and remove items from lists.
- Have students continue practicing adding and removing values from `weekend_dates`.

### 1.4.3 3. Lab

- Students will create a dictionary representing a weekly to-do list.
- The user can add items to the to-do list or have the program print what items must be done on a certain day.

### 1.4.4 4. Debrief

- Discuss any confusion or problem areas the students faced.
- Talk about how `in` works for dictionaries, specifically for the bonus.

## 1.5 Accommodation/Differentiation

---

Students that are moving quickly should work on the bonus activity in the lab. They can also be paired with students that are struggling (with close monitoring, as needed), as this lab requires pulling together a variety of concepts and skills: looping, conditionals, lists, and dictionaries.

## 1.6 Forum discussion

---

Lesson 6.03: Dictionaries of Lists (TEALS Discourse Account Required)

formatted by Markdeep 1.093 

# TEALS Program

[Home](#) | [Curriculum Map](#) | [Additional Readings](#) | [Change Log](#)

## 1 Do Now 6.03

---

### 1.1 1. In your Console

---

#### 1.1.1 Type the following

```
weekend_dates = {
 'April 2018': [7, 8, 14, 15, 21, 22, 28, 29],
 'May 2018': [5, 6, 12, 13, 19, 20, 26, 27]
}

print(weekend_dates['April 2018'])
```



### 1.2 In your Notebook

---

#### 1.2.1 Respond to the following

- What type is weekend\_dates?
- What type the are keys?
- What type the are values?

### 1.3 2. In your Console

---

#### 1.3.1 Challenge yourself with the following

- The 2018 Memorial Day holiday is observed on Monday, May 28, which makes it a long-weekend day.
- Write a new line of code that adds May 28th to the list associated with 'May 2018'
- So that its value becomes [5, 6, 12, 13, 19, 20, 26, 27, 28].

# TEALS Program

[Home](#) | [Curriculum Map](#) | [Additional Readings](#) | [Change Log](#)

## 1 Lab 6.03 - Dictionaries Storing Lists

---

In this lab your job is to create a week-long to-do list using a Python dictionary. Each key in the dictionary is a day of the week. Each associated value is a list of items to do that day.

The program repeatedly asks the user what action they wish to take (**add** or **get**).

- If the user enters **get**, the program asks for a day of the week, and then returns the to-do list for that day.
- If the user enters **add**, the program asks for a day of the week, then asks for a new item, then adds it to the specified list.
- If a user tries to add an item that already exists on the list for that day, the program rejects the request.
- At the start of the program the dictionary should be totally empty (containing no keys and no values).

### 1.1 Example

---

Here's an example. The program output is shown in bold text, the user input in regular text.

```
>>>python3 daily_to_do_list.py
What would you like to do ('add' or 'get')?
add
What day?
Friday
What would you like to add to Fridays to-do list?
practice clarinet
What would you like to do ('add' or 'get')?
get
What day?
Friday
You have to practice clarinet.
What would you like to do('add' or 'get')? //
```

### 1.2 Bonus

---

It's a bit tedious for the user to have to type in three different lines to add an item to a to-do list. Use `split()` to allow the user to input add Friday watch tv and relax.

Create a variation of the program that doesn't allow any duplicates across any of the days. Make sure when you add a to-do item that it doesn't exist in the to-do lists of any of the days before adding.





# TEALS Program

[Home](#) | [Curriculum Map](#) | [Additional Readings](#) | [Change Log](#)

## 1 Lab 6.04 - Dictionaries Looping

---

In this lab we will use our word-counting code from Lab 6.02 to create a program that determines the top 5 most commonly used words in a passage of text. After processing the passage, it prints the top 5 words and the number of times each occurs.

Here's one strategy for completing this lab:

1. Repackage some of your code from Lab 6.02 to make two functions: `text_to_word_list()`, that takes a single passage of text and splits into a list of words; and `count_frequencies()`, that takes in a list of words and returns a dictionary of word frequencies
2. Write a new function, `find_max_valued_key()`, that takes a dictionary as an argument, and returns the **key** that is associated with the largest value in that dictionary. Internally, this function loops through the dictionary while keeping track of the largest value it's seen so far and the key that goes along with that value.
3. Run `find_max_valued_key()` once on the dictionary of word counts, print out the key/value of word it returns.
4. Remove that key from the dictionary.
5. Repeat steps 3-4 four more times: Call `find_max_valued_key()`, print out the key/value pair, and remove the key.

If there is a tie within `find_max_valued_key()`, choose among the tied items however you like and return just one of them.

### 1.1 Example

---

Here's an example of the program output with the text passage set to the opening lines of Dr. Seuss's poem *Green Eggs and Ham*:

I am Sam. I am Sam. Sam-I-am.

That Sam-I-am! That Sam-I-am!  
I do **not** like that Sam-I-am!

Would you like green eggs **and** ham?

I do **not** like them, Sam-I-am.  
I do **not** like green eggs **and** ham.

Would you like them here **or** there?

I would **not** like them here **or** there.  
I would **not** like them anywhere.  
I do **not** like green eggs **and** ham.  
I do **not** like them, Sam-I-am.

Would you like them **in** a house?  
Would you like them **with** a mouse?

I do **not** like them **in** a house.  
I do **not** like them **with** a mouse.  
I do **not** like them here **or** there.  
I do **not** like them anywhere.  
I do **not** like green eggs **and** ham.  
I do **not** like them, Sam-I-am.



```
>>> python3 most_frequent_words.py
i, 22
like, 17
not, 13
do, 11
them, 12
```

//

## 1.2 Bonus

---

The process of finding the largest element, printing it, and removing it from the dictionary is a way to sort items. Write a function that will return a sorted list of all the words from most frequent to least frequent.

Change the code to find the least frequent words.

formatted by [Markdeep 1.093](#) 

# TEALS Program

[Home](#) | [Curriculum Map](#) | [Additional Readings](#) | [Change Log](#)

## 1 Lesson 6.05: Dictionaries Project

---

### 1.1 Learning Objectives

---

Students will be able to...

- Use dictionaries to create the game [Guess Who](#)

### 1.2 Materials/Preparation

---

- [Do Now](#)
- [Project Spec - Buy an Umbrella](#) ([printable Spec](#)) ([editable spec](#))
- [Project Spec - Guess Who](#) ([printable project Spec](#)) ([editable project spec](#))
- Solution (access protected resources by clicking on "Additional Curriculum Materials" on the [TEALS Dashboard](#))
- Read through the do now and project spec so that you are familiar with the requirements and can assist students.
- Try creating your own variation on the Guess Who game so you are familiar with the potential challenges and bugs your students will hit.
- Review [4 Steps to Solve Any CS Problem](#)
- [Editable Grading Rubric](#)

### 1.3 Pacing Guide

---

#### 1.3.1 Day 1

Duration	Description
5 Minutes	Do Now
10 Minutes	Review
10 Minutes	Project Overview
30 Minutes	Project Planning

#### 1.3.2 Days 2-7

Duration	Description
5 Minutes	Planning/Questions
10 Minutes	Review (if necessary)
35 Minutes	Project Work
5 Minutes	Wrap up

## 1.4 Instructor's Notes - Day 1

---

### 1.4.1 1. Do Now

- Display the Do Now on the board.
- Students should take a few moments to rank which topics they found most difficult during this unit.

### 1.4.2 2. Review

- Take time to review the concepts students found most challenging during this unit.

### 1.4.3 3. Project Overview

- Go over the project spec details with the students.
- Demo a completed project.

### 1.4.4 4. Planning

- Have the students design the variables, functions, dictionaries, lists, and structure they need, using pseudo code to help them visualize.
- Have the students write out a plan /outline for how they will spend time over next few days.
- Any students that finish their plan and have it checked should begin project work.

## 1.5 Instructor Notes - Days 2-7

---

### 1.5.1 1. Planning/Questions

- Have the students write down what they want to accomplish that day and any questions they have from the previous class.

### 1.5.2 2. Review (if necessary)

- Go over any concepts or challenges that students are having.

### **1.5.3 3. Project Work**

- Students work independently to complete their projects and meet their daily goals

### **1.5.4 4. Wrap Up**

- Discuss common issues that students were having each day, provide suggestions for how to overcome those challenges.

## **1.6 Accommodation/Differentiation**

---

- Some students may need a refresher on using a while loop to control whether the game is over or not.
- Some students may also need additional assistance or scaffolding for how to randomly choose a character from the dictionary.
- For students who are looking for more of a challenge, we have added a second project called [Project Spec - Buy an Umbrella]

## **1.7 Grading**

---

[Editable Grading Rubric](#)

### **1.7.1 Objective Scoring Breakdown**

<b>Points</b>	<b>Percentage</b>	<b>Objective</b>	<b>Lesson</b>
6	20%	The Student uses dictionaries to create key-value pairs	6.01
9	30%	The Student can use dictionary methods to update, add, and remove values from a dictionary	6.02
3	8%	The Student can utilize dictionaries of different types	6.03
3	8%	The student can use loops to traverse through key/value pairs in a dictionary	6.04
5	17%	Student can decompose a problem to create a program from a brief	
5	17%	Student uses naming/ syntax conventions and comments to increase readability	
31		<b>Total Points</b>	

## **1.8 Forums link**

---





# TEALS Program

[Home](#) | [Curriculum Map](#) | [Additional Readings](#) | [Change Log](#)

## 1 Project 6: Buy an Umbrella

---

In this project, you will build a text version of a shopping site where you can browse and filter a catalog of umbrellas before buying one.

### 1.1 Overview

---

The program starts by loading an inventory of umbrella models from a text file. Afterwards, it greets the user with this message:

```
Welcome to Umbrellas Unlimited, your online market for water protection. We have over 100 umbrellas for sale. Happy shopping!
```

Use these commands to navigate our site:

(n)ext	- view the next page of items
(p)revious	- view the previous page of items
(a)dd filter	- narrow your search by adding criteria
(r)emove filter	- broaden your search by deleting criteria
(m)odify filter	- change your search criteria
(b)uy	- purchase an umbrella <b>from</b> the list shown
(q)uit	- exit our site

Then it allows the user to browse and filter the catalog of available umbrellas:

```
Showing items 1-5 of 162 items
```

- 1) Samsonite polyester compact umbrella.  
Automatic Open **and** close, Clear. **2** stars. **\$8.99**
  - 2) GustBuster plastic compact umbrella.  
Automatic Open only, Yellow. **0.5** stars. **\$9.04**
  - 3) totes plastic golf umbrella.  
Automatic Open only, Green. **4.5** stars. **\$9.84**
  - 4) GustBuster polyester standard umbrella.  
Automatic Open only, Red. **1.5** stars. **\$10.13**
  - 5) Rainlax canvas standard umbrella.  
Yellow. **3.5** stars. **\$10.16**
- (b)uy, (n)ext, (p)revious, (a)dd, (r)emove, (m)odify, (h)elp **or** (q)uit?

We are providing you with a text file that contains the [complete inventory](#) of all 100+ models carried by Umbrellas Unlimited. Your program should use a dictionary of strings to represent an umbrella and a list of such dictionaries to represent the full inventory. To manage the search criteria, your program should use a dictionary of lists that contain acceptable values for each attribute of an umbrella.

Although you may build this entire program from scratch, we strongly suggest you start with the provided scaffolding program. Either way, we encourage you to follow the implementation plan listed below.

### 1.2 User Experience

---

Here's an example interaction session:

## 1.2.1 Paging through items

(b)uy, (n)ext, (p)revious, (a)dd, (r)emove, (m)odify, (h)elp or (q)uit? <b>next</b>

Showing items 6-10 of 162 items

- 6) Rainlax polyester golf umbrella.  
Red. 1 stars. \$10.51
- 7) ShedRain plastic golf umbrella.  
Automatic Open only, Red. 2.5 stars. \$10.51
- 8) Rainlax plastic standard umbrella.  
Automatic Open only, Yellow. 3.5 stars. \$10.82
- 9) Rainlax plastic standard umbrella.  
Yellow. 0.5 stars. \$11.08
- 10) Rainlax nylon standard umbrella.  
Clear. 2 stars. \$11.48

(b)uy, (n)ext, (p)revious, (a)dd, (r)emove, (m)odify, (h)elp or (q)uit? <b>next</b>

Showing items 11-15 of 162 items

- 11) Samsonite nylon golf umbrella.  
Yellow. 4 stars. \$11.80
- 12) GustBuster plastic standard umbrella.  
Automatic Open only, Green. 4 stars. \$12.29
- 13) Rainlax nylon standard umbrella.  
Red. 3 stars. \$13.11
- 14) Rainlax canvas golf umbrella.  
Blue. 1 stars. \$13.31
- 15) GustBuster polyester standard umbrella.  
Automatic Open and close, Green. 3 stars. \$13.43

(b)uy, (n)ext, (p)revious, (a)dd, (r)emove, (m)odify, (h)elp or (q)uit? <b>next</b>

Showing items 16-20 of 162 items

- 16) GustBuster nylon compact umbrella.  
Automatic Open only, Clear. 2.5 stars. \$13.95
- 17) Rainlax nylon golf umbrella.  
Automatic Open and close, Clear. 3 stars. \$14.25
- 18) ShedRain plastic compact umbrella.  
Automatic Open only, Blue. 1.5 stars. \$14.25
- 19) Rainlax canvas standard umbrella.  
Automatic Open only, Yellow. 5 stars. \$14.31
- 20) Rainlax plastic golf umbrella.  
Black. 2.5 stars. \$15.28

(b)uy, (n)ext, (p)revious, (a)dd, (r)emove, (m)odify, (h)elp or (q)uit? <b>previous</b>

Showing items 11-15 of 162 items

- 11) Samsonite nylon golf umbrella.  
Yellow. 4 stars. \$11.80
- 12) GustBuster plastic standard umbrella.  
Automatic Open only, Green. 4 stars. \$12.29
- 13) Rainlax nylon standard umbrella.  
Red. 3 stars. \$13.11
- 14) Rainlax canvas golf umbrella.  
Blue. 1 stars. \$13.31
- 15) GustBuster polyester standard umbrella.  
Automatic Open and close, Green. 3 stars. \$13.43



## 1.2.2 Adding filters

Showing items 1-5 of 162 items

- 1) Samsonite polyester compact umbrella.  
Automatic Open and close, Clear. 2 stars. \$8.99
- 2) GustBuster plastic compact umbrella.  
Automatic Open only, Yellow. 0.5 stars. \$9.04
- 3) totes plastic golf umbrella.

- Automatic Open only, Green. **4.5** stars. **\$9.84**  
4) GustBuster polyester standard umbrella.  
Automatic Open only, Red. **1.5** stars. **\$10.13**  
5) Rainlax canvas standard umbrella.  
Yellow. **3.5** stars. **\$10.16**

(b)uy, (n)ext, (p)revious, (a)dd, (r)emove, (m)odify, (h)elp or (q)uit? <b>add</b>

Attributes:

- 1) brand
- 2) size
- 3) material
- 4) auto
- 5) color
- 6) stars
- 7) price

Which attribute would you like to filter on? --> <b>1</b>

Values:

- 1) GustBuster
- 2) NewSight
- 3) Rainlax
- 4) Samsonite
- 5) ShedRain
- 6) totes

Which value would you like to allow ?--> <b>4</b>

Current filters:

brand: Samsonite

Showing items **1-5** of **35** items

- 1) Samsonite polyester compact umbrella.  
Automatic Open **and** close, Clear. **2** stars. **\$8.99**
- 2) Samsonite nylon golf umbrella.  
Yellow. **4** stars. **\$11.80**
- 3) Samsonite polyester golf umbrella.  
Clear. **1.5** stars. **\$15.52**
- 4) Samsonite plastic compact umbrella.  
Automatic Open **and** close, Black. **3.5** stars. **\$16.46**
- 5) Samsonite canvas golf umbrella.  
Automatic Open only, Clear. **2** stars. **\$18.25**

(b)uy, (n)ext, (p)revious, (a)dd, (r)emove, (m)odify, (h)elp or (q)uit? <b>add</b>

Attributes:

- 2) size
- 3) material
- 4) auto
- 5) color
- 6) stars
- 7) price

Which attribute would you like to filter on? --> <b>3</b>

Values:

- 1) canvas
- 2) nylon
- 3) plastic
- 4) polyester

Which value would you like to allow ?--> <b>2</b>

Current filters:

brand: Samsonite

material: nylon

Showing items **1-5** of **6** items

- 1) Samsonite nylon golf umbrella.  
Yellow. **4** stars. **\$11.80**
- 2) Samsonite nylon standard umbrella.  
Blue. **3** stars. **\$23.98**
- 3) Samsonite nylon standard umbrella.  
Red. **2** stars. **\$24.10**
- 4) Samsonite nylon golf umbrella.  
Clear. **2.5** stars. **\$31.40**
- 5) Samsonite nylon standard umbrella.  
Green. **4.5** stars. **\$32.95**

(b)uy, (n)ext, (p)revious, (a)dd, (r)emove, (m)odify, (h)elp or (q)uit?

//  
Notice that the list now contains only 6 items, all Samsonite brand and all nylon materials.

### 1.2.3 Removing filters

Continuing the previous example, we can remove the brand filter:

```
(b)uy, (n)ext, (p)revious, (a)dd, (r)emove, (m)odify, (h)elp or (q)uit? remove
Filters:
 1) brand: Samsonite
 2) material: nylon
Which criterion would you like to remove? --> 1
Current filters:
material: nylon
```

Showing items 1-5 of 37 items

- 1) Rainlax nylon standard umbrella.  
Clear. 2 stars. \$11.48
- 2) Samsonite nylon golf umbrella.  
Yellow. 4 stars. \$11.80
- 3) Rainlax nylon standard umbrella.  
Red. 3 stars. \$13.11
- 4) GustBuster nylon compact umbrella.  
Automatic Open only, Clear. 2.5 stars. \$13.95
- 5) Rainlax nylon golf umbrella.  
Automatic Open and close, Clear. 3 stars. \$14.25

```
(b)uy, (n)ext, (p)revious, (a)dd, (r)emove, (m)odify, (h)elp or (q)uit?
```



### 1.2.4 Modifying filters

Sometimes you want to allow multiple values for a given attribute, say "Nylon or plastic". A user can modify their filter to allow this:

```
Current filters:
brand: Samsonite
material: nylon
```

Showing items 1-5 of 6 items

- 1) Samsonite nylon golf umbrella.  
Yellow. 4 stars. \$11.80
- 2) Samsonite nylon standard umbrella.  
Blue. 3 stars. \$23.98
- 3) Samsonite nylon standard umbrella.  
Red. 2 stars. \$24.10
- 4) Samsonite nylon golf umbrella.  
Clear. 2.5 stars. \$31.40
- 5) Samsonite nylon standard umbrella.  
Green. 4.5 stars. \$32.95

```
(b)uy, (n)ext, (p)revious, (a)dd, (r)emove, (m)odify, (h)elp or (q)uit? modify
```

```
Filters:
 1) brand: Samsonite
 2) material: nylon
Which filter would you like to modify? --> 2
```

Values:

- 1) canvas
- \* 2) nylon
- 3) plastic
- 4) polyester

Which value would you like to check/uncheck ?--> <b>3</b>

```
Current filters:
brand: Samsonite
material: nylon, plastic
```

Showing items 1-5 of 17 items

- 1) Samsonite nylon golf umbrella.  
Yellow. 4 stars. \$11.80
- 2) Samsonite plastic compact umbrella.

- Automatic Open and close, Black. 3.5 stars. \$16.46
- 3) Samsonite plastic compact umbrella.  
Automatic Open only, Yellow. 1 stars. \$23.28
- 4) Samsonite nylon standard umbrella.  
Blue. 3 stars. \$23.98
- 5) Samsonite nylon standard umbrella.  
Red. 2 stars. \$24.10

(b)uy, (n)ext, (p)revious, (a)dd, (r)emove, (m)odify, (h)elp or (q)uit?

Note that both nylon and plastic umbrellas are now present.

### 1.2.5 Completing a purchase

Finally, the user can select an item they want and purchase it:

Current filters:  
brand: Samsonite  
material: nylon, plastic

Showing items 6-10 of 17 items

- 6) Samsonite plastic standard umbrella.  
Blue. 1.5 stars. \$25.02
- 7) Samsonite nylon golf umbrella.  
Clear. 2.5 stars. \$31.40
- 8) Samsonite plastic golf umbrella.  
Yellow. 4.5 stars. \$32.85
- 9) Samsonite nylon standard umbrella.  
Green. 4.5 stars. \$32.95
- 10) Samsonite plastic compact umbrella.  
Automatic Open and close, Clear. 5 stars. \$34.15

(b)uy, (n)ext, (p)revious, (a)dd, (r)emove, (m)odify, (h)elp or (q)uit? <b>buy</b>

What item would you like to purchase?--> <b>8</b>

You have purchased a: Samsonite plastic golf umbrella.

Yellow. 4.5 stars. \$32.85.

Enjoy!

## 1.3 Data Structures

---

The key idea of this project is that you can use Python dictionaries and Python lists together in interesting combinations. We will be using simple dictionaries of strings, but also dictionaries of lists and lists of dictionaries!

### 1.3.1 Representing an Umbrella

We'll use a simple dictionary of strings to represent each umbrella. Here's an example umbrella, represented as a dictionary:

```
one_umbrella = { 'brand': 'Samsonite',
 'size': 'compact',
 'material': 'polyester',
 'auto': 'Open and close',
 'color': 'Clear',
 'stars': '2',
 'price': '8.99',
}
```

The umbrella has various attributes like brand, size, material, etc. and each has the value shown.

### 1.3.2 Representing the Umbrella Inventory

One umbrella isn't very interesting. Our online marketplace sells over 100 models of umbrellas. Your program should keep track of them as a list of dictionaries:

```
umbrellas = [umbrella0, umbrella1, umbrella2, ...]
```

Each umbrella is a dictionary. Of course, you won't want to create this list of dictionaries by hand — there's over 100 of them. Rather, you will use the computer to construct these dictionaries for you. We have provided you with a simple text file named `inventory.txt` that lists all the models that Umbrellas Unlimited sells. Here's the first few lines of that file.

```
brand,size,material,auto,color,stars,price
Samsonite,compact,polyester,Open and close,Clear,2,8.99
GustBuster,compact,plastic,Open only,Yellow,0.5,9.04
totes,golf,plastic,Open only,Green,4.5,9.84
GustBuster,standard,polyester,Open only,Red,1.5,10.13
```

This file is in a popular format named “comma-separated values”. Each item is described by a number of attributes, like its brand, size, price, etc. The first line (the “header line”) of the file names what each of these attributes is. The rest of the file is one line per item, each line containing the values of those attributes for the particular item it represents. The order of the values is consistent with the order of the attributes in the header line. For example the first non-header line here lists an umbrella whose price is \$8.99, color is Clear, and brand is Samsonite.

In the inventory file, each item is represented by a string. In the program, we will be representing the items as dictionaries. So one of the important functions you will be writing in this lab is `load_items(filename)`. In this example, the first non-header line would become the dictionary `one_umbrella` shown above.

### 1.3.3 Representing Search Filter

The structure for representing a search filter is more complex. Here's a filter that represents “A Samsonite umbrella made of nylon or plastic”:

```
my_criteria = { 'brand': ['Samsonite'],
 'material': ['nylon', 'plastic'],
 }
```

Notice that the keys are still the names of the attributes, but the values are now *lists* of acceptable descriptions of the umbrella. One curious thing about this representation is that if a key is absent, then implicitly any value is acceptable. So for example, any color umbrella could satisfy `my_criteria`.

In the example above, does `one_umbrella` satisfy `my_criteria`? No, it does not. You can see this by checking each of the attributes in the criteria. First is brand. Does `one_umbrella` have an acceptable brand? Yes. `one_umbrella['brand']` is 'Samsonite', which is in the allowed list `my_criteria['brand']`. How about material? Here it fails. `one_umbrella['material']` is 'polyester', but 'polyester' is not in the list ['nylon', 'plastic'] (which is `my_criteria['material']`).

One of the key functions you will write in this project is `filter_items(criteria, items)`, which takes a list of items (e.g. umbrellas) and returns a smaller list of just the ones that satisfy the conditions specified in the dictionary criteria.

## 1.4 Development Strategy

---

### 1.4.1 Milestones

We strongly encourage you to write your program in stages; at the end of each stage your program must run without error, and deliver an additional increment of functionality. Here's the suggested list of development milestones.

1. Program can print the welcome message and action prompt, but

can't actually take any actions. It just states "X not implemented" for any action the user tries to take. (The starter code described below gets students to this milestone "for free".)

1. Program can load the inventory of umbrella models from the text file, and print out a nice representation of the top 10 of them.

1. Program supports the actions (n)ext and (p)revious, allowing users to browse through the full inventory of umbrellas.

1. Program supports the (b)uy action, allowing users to select one of the shown models and "purchase" it (simply prints a description of the selected item and exits).

1. Temporarily hard code the filter `my_criteria` into your program as above.

Program applies that filter to the full inventory and displays just the subset that match it.

1. Program supports the (a)dd filter action, prompting user for an attribute and a value, creates a corresponding search criteria dictionary, and applies it to the inventory (instead of `my_criteria`, which you can now delete).

1. Program supports the (r)emove filter action, displaying the current attributes that are being filtered on and asking the user which one to remove.

1. Program supports the (m)odify filter action. First it prompts the user to choose one of the attributes currently being filtered on, then it prompts the user to toggle (switch from allowed to excluded or vice-versa) one of the values for that attribute.

#### 1.4.2 Starter Project

This is a lengthy project, and writing it entirely from scratch in the time available is fairly challenging. We encourage students to use [this starter project](#) and extend it as instructed to make a finished project. Here is a tour of the starter project and a list of parts that students must implement.

The starter project contains 7 files:

File	Description
<code>inventory.csv</code>	A CSV file containing descriptions of 100+ models of umbrellas
<code>main.py</code>	Contains the top-level logic of the program, including the action loop and handlers for each action
<code>items.py</code>	Functions related to representing shopping items as python dictionaries
<code>criteria.py</code>	Functions related to search criteria
<code>dictionaries.py</code>	Utilities for manipulating dictionaries
<code>pages.py</code>	Functions to help display a paginated list
<code>TEALS_utils.py</code>	General utilities for many TEALS labs

Code from 5 of these files is incorporated into `main.py` via `import` statements at the top of that program. The file `inventory.txt` is loaded at that start of the main program.

You can run the starter project as-is. It will run without error, but it won't do much. In order to make it useful, you will have to implement a number of functions. They are already defined in the various .py files, but have stub implementation that do little besides announce that they are not yet implemented.

The starter code files also contain some fully implemented functions that you do not need to change.

Here's a table listing all of the functions present in the starter code, together with an indication of whether you need to modify them or not.

Student must implement?	Function name	File
<b>Yes</b>	handle_purchase()	main.py
<b>Yes</b>	handle_add_criterion()	main.py
<b>Yes</b>	handle_remove_criterion()	main.py
<b>Yes</b>	handle_modify_criterion()	main.py
<b>Yes</b>	load_items()	items.py
<b>Yes</b>	item_to_string()	items.py
No	print_page()	items.py
<b>Yes</b>	filter_items()	criteria.py
<b>Yes</b>	criteria_to_string()	criteria.py
<b>Yes</b>	union_of_dictionaries()	dictionaries.py
No	first_index()	pages.py
No	last_index()	pages.py
No	next_page_number()	pages.py
No	previous_page_number()	pages.py
No	safe_to_integer()	TEALS_utils.py
No	get_valid_integer()	TEALS_utils.py

Of course you are free to implement additional functions that you find useful.

Students will have to decide what is the best order to implement these functions in; we hope they will be guided by the Milestones listed above.

#### 1.4.3 Bonus

You may earn bonus points on the project by trying some of these ideas:

- The current user interface and data representation are pretty poor for continuous quantities like price and number. Users typically want to add criteria like "price < \$X" or "stars > Y". Build this feature into your program.

- Introduce a “sort by” feature that can list the items by increasing / decreasing prices, alphabetically, etc.
- Write a program to generate a random assortment of umbrella models and output them as an inventory file in CSV format.
- Update your inventory generator to produce something other than umbrellas: smart phones, bicycles, cars, clothing, etc. Update your main program in a corresponding way to create a shopping site for a different product line.

formatted by [Markdeep 1.093](#) 

# TEALS Program

[Home](#) | [Curriculum Map](#) | [Additional Readings](#) | [Change Log](#)

## 1 Project 6: Guess Who

---

In Guess Who, you'll be building a text version of the classic board game. Dictionaries will be the key to this project. [Play a Sample Game](#).

### 1.1 Overview

---

1. The game should store information on at least 5 different characters.
2. Each character should have a name, gender, age, height, and hair color.
3. When the game begins, a character should be randomly selected by the computer.
4. The player can ask for 2 pieces of information about the random character, and then has to make a guess as to who was picked.

#### 1.1.1 Behavior/Commands

- list: list out all the character's names
- gender/age/height/hair: asks for a piece of information
- guess name: guess a character
- help: displays all commands
- quit: exits the game

#### 1.1.2 Implementation Details

To store and access the information you'll need to use dictionaries, which will allow for quick and direct access.

#### 1.1.3 Example Output

```
What would you like to do? list
mike:
['Male', '15', "6'1", 'Blonde']
liv:
['Female', '25', "5'11", 'Blonde']
lisa:
['Female', '15', "5'10", 'Red']
linda:
['Female', '25', "5'7", 'Brown']
bill:
['Male', '20', "5'5", 'Brown']
What would you like to do? age
20
What would you like to do? hair
Brown
What would you like to do? guess liv
You lost...
//
```

What would you like to do? gender
Female
What would you like to do? height

5'7

What would you like to do? guess linda

//

formatted by Markdeep 1.093 📄

# TEALS Program

[Home](#) | [Curriculum Map](#) | [Additional Readings](#) | [Change Log](#)

## 1 Unit 7 - Introduction to Object Oriented Programming

---

### 1.1 Essential Questions

---

- How do we use Classes and Objects in Python?
- What are methods and how are they applied?
- How do classes communicate with one another?
- How do you plan out your code when using classes and objects?"

### 1.2 Pacing Guide

---

#### 1.2.1 Timing

1 Day = 50 minute class period

Lesson	Days	Resources
7.01:	1	<a href="#">7.01 Slide Deck</a>
7.02:	1	<a href="#">7.02 Slide Deck</a>
7.03:	1	<a href="#">7.03 Slide Deck</a>
7.04:	1	<a href="#">7.04 Slide Deck</a>
7.05:	7	<a href="#">7.05 Slide Deck</a>
<b>Total Days</b>	11	
<b>Total Minutes</b>	550	

### 1.3 Key Terms

---

- Class
- Instance
- Object
- Attributes
- Self
- `__Init__`

- Method
- \_\_str\_\_
- \_\_Add\_\_
- Operating overloading
- Inheritance
- Parent Class
- Child Class
- Class Design

formatted by Markdeep 1.093 ↗

# TEALS Program

[Home](#) | [Curriculum Map](#) | [Additional Readings](#) | [Change Log](#)

## **1 Lesson 7.01: User-Defined Types (Classes)**

---

### **1.1 Learning Objectives**

---

Students will be able to...

- Define and identify: **class, instance, object, attribute**
- Create a class and instantiate
- Add attributes to an instance
- Create an embedded object
- Manipulate instances and attributes through a function

### **1.2 Materials/Preparation**

---

- [Do Now](#)
- [Example](#)
- [Lab](#) (printable lab document) ([editable lab document](#))
- [Associated Reading](#)
- Read through the do now, lesson, and lab so that you are familiar with the requirements and can assist students

### **1.3 Pacing Guide**

---

<b>Duration</b>	<b>Description</b>
5 Minutes	Do Now
10 Minutes	Lesson
35 Minutes	Lab
5 Minutes	Discussion

### **1.4 Instructor's Notes**

---

#### **1.4.1 1. Do Now**

- Display the Do Now.

- Give the students at least five minutes to think about the questions posed in the Do Now.
- This might be a good time for a think-pair-share.

## 1.4.2 2. Lesson

### 1.4.2.1 Discuss the Do Now

- Ask the students what data type they thought would be helpful.
- They probably noticed that this was going to be difficult to do concisely without some other data type.
- Acknowledge that we need something that says "I am type Pet, which has three different qualities or attributes".

### 1.4.2.2 Instruction - Class

- A **class** is a user-defined type.
- Syntax for creating a **class**:

```
class Pet:
 """Represents a pet."""
```



### 1.4.2.3 Instruction - Instantiation and Objects

- You can create an **instance** by calling the class as if it were a function. Usually the instance is named by assigning it to a variable.

```
my_pet = Pet()
```



- If you check the type of the instance it will be Pet.
- Instances are mutable, they can be changed or updated.
- An instance can also be referred to as an **object**.
- Objects form the basis for object-oriented programming.

### 1.4.2.4 Demonstration

- Show [Example] on board to demonstrate the Pet class.
- Ask students what the difference is between this and the do now.
- Have the students identify where the class is created.
- Next have the students identify where the class is instantiated.
- Ask students what they think pet.full\_name will do.
- This is a good time to explain the concept of an attribute.

### 1.4.2.5 Instruction - Attribute

- Values assigned to an instance are **attributes** of those objects. As an example:

```
class Pet:
 """Represents a pet."""
```



```
my_pet = Pet()
my_pet.type = 'dog'
my_pet.noise = 'bark'
my_pet.full_name = 'Lassie'
```

- An object can be visualized with an **object diagram**.

\* Objects can be attributes for another object. These are **embedded objects**. The following example illustrates this.

*An object diagram for a Pet object.*

```
class Pet:
 """Represents a pet."""

class Owner:
 """Represents a pet owner."""

Instantiate a pet.
my_pet = Pet()
my_pet.type = 'unicorn'
my_pet.noise = 'neigh'
my_pet.full_name = 'Rainbow'
my_pet.owner = Owner()
me.owner.full_name = 'Princess Firebolt'
```



- An object diagram for an embedded object looks like the following.

*An object diagram with an embedded object.*

#### 1.4.2.6 Instruction - Debugging objects

- Objects have their own unique bugs that can arise.
  - Declaring a class without any code in the body (even if that code is the docstring, which is good practice anyhow), will throw a syntax error.
  - Calling an attribute that hasn't been defined will throw an attribute error.

#### 1.4.3 3. Lab

- Have the students work on the lab described in the handout.
- In the lab, the students will create classes, objects, attributes and functions that take objects as arguments.

#### 1.4.4 4. Debrief

- Write down any questions you still have about the new terms you learned today?
- Is there anything that needs more clarification?

### 1.5 Accommodation/Differentiation

---

Given the big leap taken today with much new terminology and challenging concepts, it's quite possible students will need additional class time to digest the information and finish the lab.

For students that are quickly picking up the concepts, have them create their own unique class or have them help explain to struggling students with their own examples what a class, object, instance, or attribute is.

### 1.6 Forum discussion

---

[Lesson 7.01: User-Defined Types \(Classes\) \(TEALS Discourse Account Required\)](#)



# TEALS Program

[Home](#) | [Curriculum Map](#) | [Additional Readings](#) | [Change Log](#)

## 1 Do Now 7.01

---

### 1.1 Read through the following code

---

```
my_pet_1 = 'pet'
my_pet_1_type = 'cat'
my_pet_1_noise = 'meow'
my_pet_1_full_name = 'Snuffles McGruff'

my_pet_2 = 'pet'
my_pet_2_type = 'cat'
my_pet_2_noise = 'meow'
my_pet_2_full_name = 'Snowpounce Flury'

my_pet_3 = 'pet'
my_pet_3_type = 'cat'
my_pet_3_noise = 'meow'
my_pet_3_full_name = 'Snickers Snorkel'

my_pets = [my_pet_1, my_pet_2, my_pet_3]
for pet in my_pets:
 ## print full name of each pet
```



### 1.2 In your Notebook

---

#### 1.2.1 Respond to the following

1. Write a quick description of how you would print out each of the pet's names.
2. Write down some other data structures you could use to make this easier.

# TEALS Program

[Home](#) | [Curriculum Map](#) | [Additional Readings](#) | [Change Log](#)

## 1 Example

---

### 1.1 Read the following code

---

```
class Pet:
 """Represents a pet."""

my_pet_1 = Pet()
my_pet_1.type = 'cat'
my_pet_1.noise = 'meow'
my_pet_1.full_name = 'Snuffles McGruff'

my_pet_2 = Pet()
my_pet_2.type = 'cat'
my_pet_2.noise = 'meow'
my_pet_2.full_name = 'Snowpounce Flury'

my_pet_3 = Pet()
my_pet_3.type = 'cat'
my_pet_3.noise = 'meow'
my_pet_3.full_name = 'Snickers Snorkel'

my_pets = [my_pet_1, my_pet_2, my_pet_3]
for pet in my_pets:
 print(pet.full_name)
```



formatted by [Markdeep 1.093](#)

# TEALS Program

[Home](#) | [Curriculum Map](#) | [Additional Readings](#) | [Change Log](#)

## 1 Lab 7.01: Create a Color Class

---

In this lab we will create a class that will represent colors and build a function to combine two colors.

### 1.1 Background

---

RGB is a way of storing color data. R stands for red, G stands for green, and B stands for blue. Each color is given a value from 0 to 255.

You can use this tool to see the RGB values for different colors:  
[https://www.rapidtables.com/web/color/RGB\\_Color.html](https://www.rapidtables.com/web/color/RGB_Color.html)

<https://media.24ways.org/2009/01/f1.gif>

### 1.2 Lab

---

1. Create a class, Color.
2. Instantiate at least 3 colors.
3. Add attributes of r, g, and b to those instances.
4. Create a function, add\_color, which takes in two colors and returns a color that is the sum of the two reds, greens, and blues. **Don't forget: the maximum value for R, G, or B is 255.**

formatted by [Markdeep 1.093](#) 

# TEALS Program

[Home](#) | [Curriculum Map](#) | [Additional Readings](#) | [Change Log](#)

## 1 Lesson 7.02: User-Defined Types, Part 2

---

### 1.1 Learning Objectives

---

Students will be able to...

- Define and identify: `self`, `__init__`
- Create a class with an `init` method
- Understand and use the `self` argument
- Instantiate a class with arguments

### 1.2 Materials/Preparation

---

- [Do Now](#)
- [Lab - Pet Class \(printable lab document\)](#) ([editable lab document](#))
- [Associated Reading](#)
- Read through the do now, lesson, and lab so that you are familiar with the requirements and can assist students

### 1.3 Pacing Guide

---

Duration	Description
5 Minutes	Do Now
10 Minutes	Lesson
35 Minutes	Lab
5 Minutes	Debrief

### 1.4 Instructor's Notes

---

#### 1.4.1 1. Do Now

- Display the Do Now on the board.
- Students use the `__init__` method and explore how to initialize new objects with one or more arguments.

## 1.4.2 2. Lesson

### 1.4.2.1 Discuss the Do Now

- Ask Students the following
  - What is the name of the class?
  - Where does the Pet object get instantiated?

### 1.4.2.2 Instruction - `__init__`

- `__init__` is a special function that is called when the class is first initialized.
- If there is a print statement added to the `__init__` method, when would it get printed?

### 1.4.2.3 Instruction - `self`

- `self` is a way of referring to the instance within a function.
- What does `self.name` do and how does that relate to what `my_pet.name` does?
- Previously we had added attributes *after* the class was instantiated, but `self` allows for us to assign those at once in a single method.

## 1.4.3 3. Lab

- Students will make a Pet class. Each pet will have an animal type, color, food, noise, and name.
- Next students write a function that will take a list of pets and print out their name and the food they like to eat.

## 1.4.4 4. Debrief

- Check for student understanding and completion of the lab.
- Review the two key concepts introduced today (`self` and `__init__`)

## 1.5 Accommodation/Differentiation

---

As with the previous lesson, be prepared for some students to struggle with the new vocabulary and concepts being introduced. Reinforce the use of vocabulary such as **Instantiate**, **object**, and **attribute** so that students are consistently exposed to how to use these terms.

## 1.6 Forum discussion

---

Lesson 7.02: User-Defined Types, Part 2 (TEALS Discourse Account Required)

# TEALS Program

[Home](#) | [Curriculum Map](#) | [Additional Readings](#) | [Change Log](#)

## 1 Do Now 7.02

---

### 1.1 In your Console

---

#### 1.1.1 Type the following

```
class Pet:
 def __init__(self, name):
 self.name = name

my_pet = Pet('Peter')
print(my_pet.name) //
```

### 1.2 In your Notebook

---

#### 1.2.1 Respond to the following

1. What is the purpose of the `__init__` method?
2. What if you wanted to initialize all pet objects with a name and a color?
3. How would you modify the code to create a pet object with a name of "Peter" and a color of "brown"?

# TEALS Program

[Home](#) | [Curriculum Map](#) | [Additional Readings](#) | [Change Log](#)

## 1 Lab 7.02 - Pet Class

---

In this lab, we will create a Pet class that will keep track of the type of animal, color, food, noise and name of a given animal.

### 1.1 Create a class called Pet that has the following attributes

---

- Animal (e.g., dog, cat, fish)
- Color (e.g., spotted, tabby, gold)
- Food (e.g., kibbles, tuna, fish flakes)
- Noise (e.g., meow, woof, splash)
- Name (e.g., Scooby Doo, Fluffy, Bubbles)

### 1.2 Specifications

---

1. Make sure to use the `__init__` method to create these attributes.
2. Create a list of pets.
3. Create a function that takes in a list of pets and prints out the name and the food attributes.
4. Test your function with your list of pets.

formatted by [Markdeep 1.093](#) 

# TEALS Program

[Home](#) | [Curriculum Map](#) | [Additional Readings](#) | [Change Log](#)

## 1 Lesson 7.03: Methods

---

### 1.1 Learning Objectives

---

Students will be able to...

- Define and identify: **method**, **`__str__`**, **`__add__`**, **operator overloading**
- Create a class with an `__init__` method
- Understand and use the `self` argument
- Instantiate a class with an argument

### 1.2 Materials/Preparation

---

- [Do Now](#)
- [Lab - Kangaroo Class \(printable lab document\)](#) ([editable lab document](#))
- [Associated Reading](#)
- Read through the do now, lesson, and lab so that you are familiar with the requirements and can assist students

### 1.3 Pacing Guide

---

Duration	Description
5 Minutes	Do Now
10 Minutes	Lesson
35 Minutes	Lab
5 Minutes	Debrief

### 1.4 Instructor's Notes

---

#### 1.4.1 1. Do Now

- Display the Do Now on the board.
- Students will find that when they try to print the two different Time objects, it produces output that's not particularly useful or readable.

- Students will also discover that adding objects doesn't work...yet!

## 1.4.2 2. Lesson

### 1.4.2.1 Instruction - Method

- a function inside of a class.
- The first argument is always `self`.

### 1.4.2.2 Discussion

- Ask students what method we have already seen and used previously. (`init`)
- Ask students how they would distinguish between the two time variables.

### 1.4.2.3 Instruction - `__str__`

- Need a method called `__str__`.
- This will get called when you print an object
- it returns a string that is easy to read and understand

### 1.4.2.4 Activity

- Have the students practice writing `__str__` for the `Time` class for 5 minutes.
- Have a student write up their string method on the board.

### 1.4.2.5 Instruction `__add__`

- A method that gets called when the plus sign is used between two `Time` objects.
- In this case it takes as parameters `self` and another `Time` object and returns a `Time` object that is the sum of both.
- Overwriting `add` is called **operator overloading** because you are re-writing the code used to make the `+` work.

### 1.4.2.6 Demonstration

- Work together with students to come up with the add time algorithm.

## 1.4.3 3. Lab

- Have students finish up the time adding method.
- Have students work on kangaroo lab.

## 1.4.4 4. Debrief

- Go over students' questions and demonstrate some students' successfully completed labs.
- Review what a method is, as well as what specific methods were used in today's lab.

## **1.5 Accommodation/Differentiation**

---

Students that are moving quickly should work on the bonus assignment in the lab or assist a partner that is struggling.

## **1.6 Forum discussion**

---

Lesson 7.03: Methods (TEALS Discourse Account Required)

*formatted by Markdeep\_1.093* 



# TEALS Program

[Home](#) | [Curriculum Map](#) | [Additional Readings](#) | [Change Log](#)

## 1 Lab 7.03 - Kangaroo Class

---

### 1.1 Instructions

---

- Finish writing the `__add__` method for the time class from the Do Now.
- Write a definition for a class named Kangaroo with the following methods:
  - An `__init__` method that initializes an attribute named `pouch_contents` to an empty list.
  - A method named `put_in_pouch` that takes an object of any type and adds it to `pouch_contents`.
  - A `__str__` method that returns a string representation of the Kangaroo object and the contents of the pouch.

#### 1.1.1 Tips to give students

- This exercise is a cautionary tale about one of the most common, and difficult to find, errors in Python
- `TypeError`: Can't convert 'list' object to `str` implicitly
- Use the `str()` function to convert the list object to a string.
- Test your code by creating two Kangaroo objects
  - assign them to variables named `kanga` and `roo`
  - add `roo` to the contents of `kanga`'s pouch

### 1.2 Extra Credit

---

Return to your Pet class from Lab 7.02. Research the `isinstance` function to write a method, `is_friend` that will take in another pet and return `True` if the two pets are friends, and `false` if they are not.

#### 1.2.1 Rules

- If they are both dogs they are friends.
- If the instance is a dog and the other pet is a cat, they are friends.
- If the instance is a cat and the other is a dog they are not friends.
- If they are both cats they are not friends.





# TEALS Program

[Home](#) | [Curriculum Map](#) | [Additional Readings](#) | [Change Log](#)

## 1 Lab 7.04 - Pokemon Child Classes

---

### 1.1 Overview

---

Given the following [example], practice using inheritance to create specific child classes for different types of Pokemon.

#### 1.1.1 Create the three child classes below

##### 1.1.1.1 1. Water Type

- When attacking a fire type, the attack is more effective
- When attacking a grass type the effect is less effective
- When growl is called print out Splash

##### 1.1.1.2 2. Fire Type

- When attacking a water type, the attack is less effective
- When attacking a grass type the effect is more effective
- When growl is called print out "Fire Fire"

##### 1.1.1.3 3. Grass Type

- When attacking a water type, the attack is more effective
- When attacking a fire type the effect is less effective
- When growl is called print out "Cheep Cheep"

**Note:** In order to check what type an object is you can use `isinstance` which takes in an object, a class and returns a Boolean if the object is the type of the inputted class.

### 1.1.2 Example Code

```
my_pet = Pet()
isinstance(my_pet, Pet) # returns true
isinstance(my_pet, Dog) # returns false
```





# TEALS Program

[Home](#) | [Curriculum Map](#) | [Additional Readings](#) | [Change Log](#)

## 1 Lesson 7.05: Pokemon Project

---

### 1.1 Learning Objectives

---

Students will be able to...

- Engage in **class design** before beginning coding
- Apply what was learned with respect to **classes**, **methods**, and\*\* inheritance\*\* to create an implementation of Pokemon

### 1.2 Materials/Preparation

---

- Project Spec - Pokemon ([printable project Spec](#)) ([editable project spec](#))
- Project Spec - Mailing List ([printable alternate project Spec](#)) ([editable alternate project spec](#))
- Solution (access protected resources by clicking on "Additional Curriculum Materials" on the [TEALS Dashboard](#))
- Read through the project spec so that you are familiar with the requirements and can assist students
- Try creating your own project so that you can Review [4 Steps to Solve Any CS Problem](#)
- [Editable Grading Rubric](#)

#### 1.2.1 Day 1 Pacing

Duration	Description
5 Minutes	Project Handout
5 Minutes	Mini-Lesson
15 Minutes	Project Overview
30 Minutes	Project Planning

#### 1.2.2 Days 2-7 Pacing

Duration	Description
5 Minutes	Planning/Questions
10 Minutes	Review
35 Minutes	Project Work
5 Minutes	Wrap up

## 1.3 Instructor's Notes

---

### 1.3.1 Day 1

#### 1.3.1.1 1. Handout Project Specifications

- Read through the Project Spec with students
- Demo a completed project to show user experience.

#### 1.3.1.2 2. Mini-Lesson

- Discuss **Class Design**
- If you find yourself creating many classes with similar methods, use inheritance!
- Figure out the actual structure without writing code and use that to create your classes

#### 1.3.1.3 3. Project Overview

- Go over the Pokemon project spec
- Review the major aspects and requirements of the game

#### 1.3.1.4 4. Project Planning

- Have students write down the classes and methods they need to create
- Students should then outline what they will do each day in order to complete the project on time

### 1.3.2 Days 2-7

#### 1.3.2.1 1. Planning/Questions

- Have students review and update what they want to accomplish that day and any questions they have from the previous day.

### **1.3.2.2 2. Review**

- if necessary, review any concepts or struggles the class was having.

### **1.3.2.3 3. Project Work**

- students work on their projects independently.

### **1.3.2.4 4. Wrap Up**

- have the students write down what they struggled on or had a hard time doing.

## **1.4 Grading**

---

### **1.4.1 Scheme/Rubric**

[Editable Grading Rubric](#)

<b>Points</b>	<b>Percentage</b>	<b>Objective</b>	<b>Lesson</b>
9	32%	The Student can create a class and an instance	7.01, 7.02
6	21%	The student can create methods for classes	
3	11%	The student can correctly use inheritance	
5	18%	Student can decompose a problem to create a program from a brief	
5	18%	Student uses naming/ syntax conventions and comments to increase readability	
28		<b>Total Points</b>	

## **1.5 Forum discussion**

---

[Lesson 7.05: Pokemon Project \(TEALS Discourse Account Required\)](#)

# TEALS Program

[Home](#) | [Curriculum Map](#) | [Additional Readings](#) | [Change Log](#)

## 1 Project 7: Pokemon

---

In this project we will be creating a basic implementation of Pokemon, and then create a game for them to battle. This project will have three steps. The Pokemon Classes, the User classes, and the game loop.

### 1.1 Overview

---

#### 1.1.1 9 Pokemon

Each Pokemon comes from 1 of 3 types:

1. Grass
2. fire
3. water

#### 1.1.2 Classes

You will be given a base Pokemon class which the grass, fire, and water classes will inherit from.

#### 1.1.3 Class: User

- You will be given a bare-bones implementation of the base user class.

#### 1.1.4 Class: Computer

- The computer class should inherit from the user class
- The computer class will need to overwrite certain methods of the user class.

#### 1.1.5 Attacks

All Pokemon of the same type have the same set of attacks, one of the following:

1. hp (hit points)
2. pp(power points)
3. name.

#### 1.1.6 Game Loop

Before the battle begins, Ask the user:

1. Name
2. Computer Name

#### **1.1.7 Battle Pairing**

1. Ask the user to pick 3 Pokemon for their hand
2. Randomly select the computer's Pokemon from the remaining Pokemon.
3. Ask the user which Pokemon they would like to use in battle with currently
4. Randomly select one for the Computer
5. Battle begins.

#### **1.1.8 Each Turn**

The user will be given the option to

1. attack
2. heal
3. switch.
4. Computer will be randomly selected, but should more frequently be attack.

#### **1.1.9 Attack**

Gives the user the options

1. attacks the Pokemon
2. pick one to attack with

#### **1.1.10 Attack Name**

An attack has a name, and corresponds to a list of the power of the attack and the accuracy.

#### **1.1.11 An Attack should**

1. be a random number in the range between the attack power(or the pp if that is lower than the attack power) - 20
2. the attack power (or the pp if that is lower than the attack power)
3. randomly fail according to the attack's accuracy.
4. decrease the enemy's hp, and print out a useful description.
5. The computer randomly selects an attack to use in the battle.

#### **1.1.12 Heal**

1. Gives 20 HP back to the current Pokemon,
2. Takes away the turn

#### **1.1.13 Switch**

1. Lists the stats (HP, AP, Name) of all Pokemon in the hand, and then
2. switches the current Pokemon

3. takes away the turn.
4. On the computer's turn if this option is randomly selected, choose a random Pokemon to switch with
5. If either the user's or the computer's current Pokemon is out of HP, they will be required to switch Pokemon
6. If there are no available Pokemon, that player has lost.

#### **1.1.14 Continue looping until one player has lost**

#### **1.1.15 Behaviour**

##### **1.1.15.1 Game Play**

- Ask the user to choose 3 Pokemon: At the beginning of the game the program will list out the pokemon to choose from. Options are the following, HP stands for the health points the pokemon has and AP says the max value of attack
- Grass Type:
  - Bulbasoar: 60HP, 40AP
  - Bellsprout: 40HP, 60AP
  - Oddish: 50HP, 50AP
- Fire Type:
  - Charmainder: 25HP, 70AP
  - Ninetales: 30HP, 50AP
  - Ponyta: 40HP, 60AP
- Water Type:
  - Squirtle: 80HP, 20AP
  - Psyduck: 70HP, 40AP
  - Polywag: 50HP, 50AP
- Ask the user to choose Pokemon to use in fight
- Generate Computer's Hand and randomly select a pokemon to start
- On each turn the player or computer can either attack, heal, or switch
- At the end of each the turn print out the HP of the current Pokemon, and check if either side has lost the game
- A player has lost if all their Pokemon are out of HP
- switch: Lists the stats (HP, AP, Name) of all Pokemon in the hand, and asks the user what Pokemon they would like to switch to. Cannot attack until next turn.
- stats: Lists all the stats of the Pokemon in the hand
- heal: adds 20 hp to the current Pokemon, cannot attack until next turn
- attack: lists all attacks of current Pokemon, prompts the user pick an attack and then inflicts that damage on the Computer's current pokemon. If this K.Os the enemy then the Computer must switch Pokemon on their next turn.

Should print out the following:

```
>>> Ash is attacking. Bulbosaur used Leaf Storm and did 20 damage.
```

//

##### **1.1.15.2 Computer**

- Computer will select between the three options. 2/3 of the time a user should attack. 1/6 of the time they should heal and 1/6 of the time they should switch pokemon.
- attack: Will randomly select an attack and do damage to the User's current Pokemon.
- heal: Will heal current Pokemon by 20 HP, but cannot attack that turn
- switch: switched current Pokemon, but cannot attack

#### **1.1.16 Pokemon Attacks**

All Pokemon of the same type have the same Attacks.

#### 1.1.16.1 Grass Type

All grass type have the following attacks:

- Leaf Storm: 130 Power, 90% accurate
- Mega Drain: 50 Power, 100% accurate
- Razor Leaf: 55 Power, 95% accurate

Grass Type is 1.5x stronger against Water Type.

#### 1.1.16.2 Fire Type

All fire type have the following attacks

- Ember: 60 Power, 100% accurate
- Fire Punch: 85 Power, 80% accurate
- Flame Wheel: 70 Power, 90% accurate

Fire Type is 1.5x stronger against Grass.

#### 1.1.16.3 Water Type

All fire type have the following attacks:

- Bubble: 40 Power, 100% accurate
- Hydro Pump: 185 Power, 30% accurate
- Surf: 70 Power, 90% accurate
- Water Type is 1.5x stronger against Fire Types.

### 1.1.17 Implementation Details

- Use classes to build off the implemented base class for Pokemon. Keep track of HP, Max AP, and type.
  - Here is some [starter\_code]
- Design classes for Fire, Grass, Water Types that inherit from the base Pokemon Class
- Each Pokemon should be an instance, use a master list to store all the Pokemon and create this the beginning of the game
- Use methods to set the Pokemon for each player and remove those pokemon from the master Pokemon list.
- Player's and Computer's Pokemon should be stored using a list
- Pokemon attacks should be stored using a dictionary from the attack name to a list of [attack's power, attack's accuracy]

### 1.1.18 Check Point 1

Implement the following:

- get\_attack\_power on Pokemon class
- attack on Pokemon class
- take\_damage on Pokemon class
- heal on Pokemon class
- Grass, Fire, and Water type classes
- set\_type on Fire, Water and Grass classes
- set\_attacks on Fire, Water and Grass classes

- get\_attack\_power on Fire, Water, and Grass classes

### 1.1.19 Check Point 2

Implement the following:

- list\_pokemon on User Class
- switch on User Class
- heal on User Class
- is\_end\_game on User Class
- print\_attacks on USer Class
- attack on User Class

#### 1.1.19.1 Computer class

- play\_turn on Computer class
- set\_pokemon on Computer Class
- attack on Computer Class
- switch on Computer Class

### 1.1.20 Check Point 3

Implement the following

- a game loop
- inputs that ask if user would like to attack, heal or switch
- call correct player function based on inputs
- check for the end of the game and end game if necessary

formatted by Markdeep 1.093 

# TEALS Program

[Home](#) | [Curriculum Map](#) | [Additional Readings](#) | [Change Log](#)

## 1 Alternate Project 7: Mailing List

---

Created by Brian Weinfeld

In this project, you will use dictionaries, lists and objects with Python to create a program that generates mailing lists for advertisers.

### 1.1 Overview

---

Every company stores data on its users and a common use of this data is to send relevant advertisements. If the company can track your interests, they can email you when they are selling something they think you may want to purchase. This project consists of several common functions designed to identify which customers you want to send advertisements to.

### 1.2 Details

---

#### 1.2.1 Behavior

```
What would you like to do? (add, count, send, display, exit) add
Enter name email and hobbies: Alice a_person@host.com horses,traveling
a_person@host.com is not in our list
a_person@host.com has been added to our list
```

```
What would you like to do? (add, count, send, display, exit) add
Enter name email and hobbies: Blake myemail@host.com school
myemail@host.com is not in our list
myemail@host.com has been added to our list
```

```
What would you like to do? (add, count, send, display, exit) display
a_person@host.com Alice ['horses', 'traveling']
myemail@host.com Blake ['school']
```

```
What would you like to do? (add, count, send, display, exit) add
Enter name email and hobbies: Alice a_person@host.com sleeping
a_person@host.com is already in our list
Added sleeping to a_person@host.com' hobbies
```

```
What would you like to do? (add, count, send, display, exit) display
a_person@host.com Alice ['horses', 'traveling', 'sleeping']
myemail@host.com Blake ['school']
```

```
What would you like to do? (add, count, send, display, exit) count
Hobby Results
{'horses': 1, 'traveling': 1, 'sleeping': 1, 'school': 1}
```

```
What would you like to do? (add, count, send, display, exit) send
Which hobby? horses
Mailing horses to: ['a_person@host.com']
```

```
What would you like to do? (add, count, send, display, exit) exit
Goodbye
```



## 1.2.2 Implementation Details

- Begin by completing the Person class. The person class represents each person your company is tracking. **name** represents the person's name, **email** represents their email, and **hobbies** is a list of strings representing the things you know this person is interested in. The list may be empty but there should never be repeats (you don't want the same hobby for the same person listed twice). While it is possible for many people to have the same **name** and **hobbies**, we will assume that each **email** is unique. That is, only one person can have a specific email address.

```
class Person:

 def __init__(self, name, email, hobbies):
 # finish

 def __str__(self):
 # finish
```

- Next, complete the 4 methods in the Mailer class. These will be used to create and interact with a list of people. They are described in more detail below.

```
class Mailer:

 def __init__(self):
 self.people = []

 def __str__(self):
 return '\n'.join(str(p) for p in self.people)

 def send_hobby_mailer(self, hobby):
 # finish

 def count_hobbies(self):
 # finish

 def already_present(self, check):
 # finish

 def add_person(self, check):
 # finish
```

- In **send\_hobby\_mailer** create a list of emails that contain all the people who you know are interested in that hobby. Print the list.
- In **count\_hobbies** print each hobby and the number of people who have it as a hobby. This will let you determine which products your company should carry.
- **already\_present** has one parameter check that is of type Person. This function determines whether this person is already in our mailing list. If they are not in our mailing list, print a message and return None. If they are in our mailing list, print a message and return the person.
- **add\_person** has one parameter check that is of type Person. This function does one of two actions. If the person is not in the mailing list, add them to the mailing list. If the person is already in the mailing list, we don't want to add them again. Instead, add the hobbies listed to their already identified hobbies to create a new, possibly longer list of hobbies. Be careful to not add any hobbies that are already in the list.

Below is an example of the functionality of the four functions.

```
> mailer = Mailer()
> mailer.add_person(Person('Alice', 'a_dog@host.com', ['dogs', 'animals']))

a_dog@host.com is not in our list
a_dog@host.com has been added to our list

> mailer.add_person(Person('Bob', 'knitting@host.com', ['knitting', 'surfing', 'painting']))

knitting@host.com is not in our list
knitting@host.com has been added to our list

> mailer.add_person(Person('Carlos', 'filmbuff@host.com', ['movies']))
```

```
filmbuff@host.com is not in our list
filmbuff@host.com has been added to our list
```

```
> mailer.add_person(Person('Daisy', 'soccerfan@host.com', ['soccer', 'tennis', 'dogs']))
```

```
soccerfan@host.com is not in our list
soccerfan@host.com has been added to our list
```

```
> mailer.add_person(Person('Eva', 'everything@host.com', ['surfing', 'movies', 'knitting', 'animals']))
```

```
everything@host.com is not in our list
everything@host.com has been added to our list
```

```
> print(mailer)
```

```
a_dog@host.com Alice ['dogs', 'animals']
knitting@host.com Bob ['knitting', 'surfing', 'painting']
filmbuff@host.com Carlos ['movies']
soccerfan@host.com Daisy ['soccer', 'tennis', 'dogs']
everything@host.com Eva ['surfing', 'movies', 'knitting', 'animals']
```

```
> mailer.add_person(Person('Bob', 'another_bob@host.com', ['cats', 'tennis']))
```

```
another_bob@host.com is not in our list
another_bob@host.com has been added to our list
```

```
> mailer.add_person(Person('Bob', 'knitting@host.com', ['surfing', 'animals', 'poetry']))
```

```
knitting@host.com is already in our list
Added animals to knitting@host.com' hobbies
Added poetry to knitting@host.com' hobbies
```

```
> print(mailer)
```

```
a_dog@host.com Alice ['dogs', 'animals']
knitting@host.com Bob ['knitting', 'surfing', 'painting', 'animals', 'poetry']
filmbuff@host.com Carlos ['movies']
soccerfan@host.com Daisy ['soccer', 'tennis', 'dogs']
everything@host.com Eva ['surfing', 'movies', 'knitting', 'animals']
another_bob@host.com Bob ['cats', 'tennis']
```

```
> mailer.count_hobbies()
```

```
Hobby Results
```

```
{'dogs': 2, 'animals': 3, 'knitting': 2, 'surfing': 2, 'painting': 1,
'poetry': 1, 'movies': 2, 'soccer': 1, 'tennis': 2, 'cats': 1}
```

```
> mailer.send_hobby_mailer('animals')
```

```
Mailing animals to: ['a_dog@host.com', 'knitting@host.com', 'everything@host.com']
```

```
> mailer.send_hobby_mailer('tennis')
```

```
Mailing tennis to: ['soccerfan@host.com', 'another_bob@host.com']
```

```
> mailer.send_hobby_mailer('unique')
```

```
Mailing unique to: []
```



- Finally, create a loop that allows a user to enter commands **add**, **count**, **send**, **display**, **exit** to interact with the code you have already created.

### 1.2.3 Challenge

This section contains additional components you can add to the project. These should only be attempted after the project has been completed.

- As in previous projects, change the looping code so that all typed information can be done at once, instead of spread over multiple lines

- Create a new function **send\_hobby\_mailer\_any**. The hobby parameter in this function is a list of hobbies. Create a list of emails that includes a person if they enjoy ANY of the hobbies listed. Modify the loop to allow this option to be selected.
- Create a new function **send\_hobby\_mailer\_all**. The hobby parameter in this function is a list of hobbies. Create a list of emails that includes a person if they enjoy ALL of the hobbies listed. Modify the loop to allow this option to be selected.

#### 1.2.4 Super Challenge

The super challenge will require knowledge that has not been taught yet. You will need to do additional research on your own. Good luck!

A common problem that data engineers face is unclean data. That is, data that is not perfectly entered by the user. For example, capitalization does not matter in an email address. [My\\_email@host.com](#) and [my\\_email@host.com](#) will go to the same place. Punctuation like periods also does not matter. [MyEmail@host.com](#) and [My.Email@host.com](#) will go to the same place. Right now, your code cannot tell the difference between these two and will treat them as different people.

Part of a data engineer's job is to clean all data inputs. Clean all of the emails in your program as they are entered. Standardize how they are stored so that you can now tell if unclean data entrees like those above are actually the same address. What other types of mistakes might a user make? Try to fix those as well!

formatted by [Markdeep 1.093](#) 

# TEALS Program

[Home](#) | [Curriculum Map](#) | [Additional Readings](#) | [Change Log](#)

## 1 Unit 8 - Final Project

---

### 1.1 Essential Questions

---

- How do you design plan and execute a medium to large scale project using Python?
- How do you write specifications for a medium to large scale project using Python?"

### 1.2 Pacing Guide

---

#### 1.2.1 Timing

1 Day = 50 minute class period

Lesson	Days
8.01:	1
8.02:	1
8.03:	1
8.04:	15
<b>Total Days</b>	<b>18</b>
<b>Total Minutes</b>	<b>900</b>

### 1.3 Grading Scheme/Rubric

---

[Editable Grading Rubric](#)

<b>Design Phases</b>	<b>Points</b>
Brainstorming	2
Project Pitches	6
Scenario Definition	4
Flow Chart	4
Project Organizer (Specification)	8
Implementation Plan	8
Spec and plan are updated throughout project	8
<b>Subtotal</b>	<b>40</b>
<b>Implementation</b>	
Project is appropriately complex and creative	8
Program is well-documented and shows good style	4
Program uses Python elements effectively, including all required elements	8
Final product meets all requirements and goals laid out in spec	8
Checkpoint 1	4
Checkpoint 2	4
Checkpoint 3	4
<b>Subtotal</b>	<b>40</b>
<b>Total</b>	<b>80</b>

formatted by [Markdeep 1.093](#) 

# TEALS Program

[Home](#) | [Curriculum Map](#) | [Additional Readings](#) | [Change Log](#)

## 1 Project- Final Project

---

- [printable project spec](#)
- [editable project spec](#)

Students will design, plan, and implement a medium- to large-scale final project of their own choosing.

### 1.1 Overview

---

During this course, you have learned a huge amount about computer science and programming in general, and Python in particular. In this project, you will put all of that knowledge, along with some new skills you will develop around design, planning, and project management, to build a relatively large and complex application that *you* choose. You can create almost anything you want and should ultimately produce a project that is interesting, useful, and challenging.

### 1.2 Details

---

#### 1.2.1 1. Project Phases

This project will be significantly larger in scope than any of your previous assignments, so there will be more design and planning than before. More importantly, though, rather than be given a well-defined specification, *YOU* will be setting the requirements for your project by coming up with an idea, fleshing out the details, and defining the steps necessary to complete your program.

To help you through this process, there will be several steps to this project. You must complete **all** of the steps **in order** for your project to be successful. In fact, *half* of your grade will be based not on how well your program works, but on how well you completed the design and planning process.

The phases of the project will be:

1. *Brainstorming* - coming up with as many possible project ideas as you can
2. *Pitching* - choosing a few ideas and developing a short description of what the project will entail
3. *Review* - getting feedback from your peers and instructors on your pitches and choosing one
4. *Scenario Definition* - listing out the features the project will need and what they will look like
5. *Flow Chart* - drawing simple flow chart of what the various interactions in your program will look like
6. *Specification* - fleshing out all the specifics of how the project will work
7. *Scheduling* - listing the programming tasks necessary to complete your project and estimating how long each will take
8. *Development* - writing the code for your project by following the spec and schedule created in the previous steps

#### 1.2.2 2. Progress Tracking

In phase vi, you will complete a Final Project Spec and in phase vii you will complete a Final Project Schedule. These documents will be your guides in the development phase and will help you stay on track and aware of

your progress. Throughout the development phase of the project, you will be expected to keep your spec and plan up-to-date and make adjustments as you get ahead or behind, as requirements change, or as tasks or features get re-prioritized. At the end of each coding day, your spec and plan documents should be updated to reflect the current state of your project, and you will check in with an instructor at least once a week to make sure things are on track.

### **1.2.3 3. Implementation Requirements**

#### **1.2.3.1 Complexity and Creativity**

Your final project should be sufficiently complex and large-scale to push your limits as a programmer, but not so sophisticated that you are not able to complete it in the time allotted. The complexity in your project should come from the *design* and the *algorithms* and not from the *code*. (That is, you cannot meet the complexity requirement simply by writing a lot of code. Your code must be challenging or interesting in some meaningful way.) In addition, you should not add complexity by introducing peripheral elements, such as graphics or sound effects. (Your program can certainly have these, but they will not be considered in determining the projects complexity.)

In addition, one of the main goals of this project is to allow you to unleash your creativity and allow you to create something of interest to you. To achieve this, your project must show some level of creativity or personalization that makes it your own. Simply creating your own version of some existing application will not fully meet this requirement.

For both the complexity and creativity requirements, you should talk to the instructors early and often to ensure your project is in line with our expectations.

#### **1.2.3.2 Documentation and Style**

As with all previous projects, your program must be well-written, well-documented, and readable. Writing code with good style is always a good idea, but in a project of this size and scope, following style guidelines will help you keep your thoughts organized and make it easier to keep track of your progress, pick up where you left off each day, and find and fix bugs. In particular, though this is certainly not a comprehensive list, pay attention to the following:

- organizing your functions/code so that they can be read and comprehended easily
- giving your functions, variables, lists, and dictionaries descriptive and meaningful names
- using the right data type (`string`, `int`, `bool`, `float`, `dictionary`, `class`) for each situation
- include comments to describe the structure of your program and track your progress
- avoiding redundancy with good use of loops, functions, and/or lists, and/or dictionaries, and/or classes
- practicing good procedural decomposition and abstraction

#### **1.2.3.3 Required Python Elements**

In order to show that you have fully mastered all the skills from the course, your project must include at least the following:

1. A clear way to start the program, and clear prompts or instructions for any user interaction
2. At least one loop, variable, function, and list, and more as necessary or appropriate
3. Each of these must be used correctly and meaningfully
  - creating a list that contains
  - a single element just to meet this requirement will not earn points
4. at least one user interaction
5. this can be prompting for information using `ask`, responding to key presses or mouse movements, or any other action that keeps the user involved

#### 1.2.3.4 Required Checkpoints

At least three times during the project period, and at least once each week, you should check in with an instructor to ensure that your project is on track, that you are meeting the project requirements, and that you have the answers to any questions that might have arisen during your work. The course staff will work with you to set up a schedule for these checkpoints, but it is **your responsibility** to ensure that the meetings take place.

formatted by [Markdeep 1.093](#) 

# TEALS Program

[Home](#) | [Curriculum Map](#) | [Additional Readings](#) | [Change Log](#)

## **1 Lesson 8.1 - Project Planning Summary + Brainstorming and Evaluating**

### **1.1 Learning Objectives**

---

Students will be able to...

- Recall project planning basics from last semester
- Identify factors to use when choosing between project ideas
- Rank a group of proposed project ideas using the identified factors

### **1.2 Materials/Preparation**

---

- Project Ideas for Students
- Editable Project Ideas
- Printable Project Ideas

### **1.3 Pacing Guide**

---

<b>Duration</b>	<b>Description</b>
5 minutes	Do Now
10 minutes	Introduce Project and Review Design
5 minutes	Brainstorming
10 minutes	Pitch writing
20 minutes	Peer review
5 minutes	Debrief and wrap-up

### **1.4 Instructor's Notes**

---

#### **1.4.1 1. Do Now**

- Project the Do Now on the board, circulate around the class to check that students are working and understand the instructions.

## 1.4.2 2. Introduce Project and Review Design

### 1.4.2.1 Discussion

- Talk about how far students have come this semester
- Ask students to think back to the start of the semester and remember how little they knew about Python programming.
- Show a lab or assignment from early on and remind them that, not that long ago, this was challenging, whereas it now seems nearly trivial (hopefully).

### 1.4.2.2 Project Overview

- Explain that, for their final project, the students will get to design and build a program of their own choosing.
- Point out that this will involve more than just writing code
- There will be planning, design, scheduling, and other project management tasks
- Emphasize that students will be graded on not only the program they produce, but the process they used to design, plan, and implement it
- Here are some [Project Ideas](#)

### 1.4.2.2.1 Demonstration

- Demonstrate a few example projects (with as much variety as possible).
- Try to hit a bunch of different types of programs.
- Many students will gravitate towards games, but other options include simulations, productivity tools, musical projects, animations, and more.

### 1.4.2.2.2 Rubric Discussion

- Distribute the project rubric and point out key aspects of the requirements
- Point out the steps in the process and that each one is equally important
- Specifically mention the large number of points for things *not* related to coding

### 1.4.2.3 Review process and identify first steps

- Display the Spec and Plan documents from last semester
- Ask students to identify the steps in the design and planning process as discussed in [do\\_now](#).
- Remind students that all steps are vital, and that thorough and thoughtful planning and design can make the coding phase much easier.
- Inform students that today they will take the first steps in designing their final project.

### 1.4.2.4 Brainstorming

- Give students 3-4 minutes to brainstorm and write down as many project ideas as they can. This should be done mostly in silence.
- At this point, there should be minimal detail, no evaluation or rejection of ideas, and no discussion. In particular, students should not think about the difficulty or “coolness” of the project yet. Just write down

- ideas.
- If desired, have each student share one idea. Do not allow discussion, criticism, or explanation— each idea should be summarized in only a few words or a single sentence.

#### 1.4.2.5 Pitch writing

- Have students pick their **3** favorite ideas and write a “pitch” for the project.
- A pitch should be no more than a short paragraph and should describe the basic, high-level features of the project.
- The pitch should *not* include any implementation details (scripts, sprites, etc.).
- Pitches should include a moderate level of specificity— enough for someone to imagine how the app will work, but not so much to get bogged down.
- Enforce the “one short paragraph” restriction.
- If a student is having difficulty developing a pitch for an idea, that might be a sign that the idea is not fully-formed enough to be a final project.
- If a student is having trouble keeping the pitch short, the project may be too complex to complete in the available time.

#### 1.4.2.6 Peer Review

- Pair students up and have students take turns reading one of their pitches to their partner and asking for feedback.
- Partners should ask questions to help identify both the best and worst parts of each pitch.
- Remind students to keep all feedback constructive, respectful, and professional.
- Students should not criticize each other's ideas, but can point out potential concerns.
- Students should take notes during their conversations and refine their pitches based on their partner's feedback and their own realizations.
- If time allows (or over the course of multiple days), repeat this process with new partners.

#### 1.4.3 3. Debrief

- At this points, students should have between one and three pitches that are well-defined and reasonably well fleshed-out.
- For Homework, students should consider their pitches and rank them in order of which they would most like to pursue as their final project.
- Make sure students don't just pick the “coolest” sounding idea, but also consider the technical challenges, amount of time available, and their own interest in and willingness to see the project through to completion.

### 1.5 Accommodation/Differentiation

---

- If students are having difficulty coming up with project ideas, encourage them to think about existing software. While simply recreating an existing app should be a last resort, thinking about apps they already know can help students come up with functionality they might like to include.
- If your class is fairly self-sufficient and mature, you can consider allowing students to “borrow” an idea from a classmate if they find one they like better than any of their own. Make sure the person who had the idea is OK with it being borrowed, and emphasize that the students must each build their own version.
- This can be a bit dangerous, as it puts the student somewhat behind in the process right from the start, since they won't have as much context having not written the pitch themselves. Consider carefully whether this is a good idea.
- Encourage each student to pick a project that fits his or her level of technical skill. Make sure students are not overcommitting and choosing projects they do not have the skills to complete. Also try to discourage stronger students from choosing simpler projects in an effort to do less work.

## 1.6 Forum discussion

---

Lesson 8.1 - Project Planning Summary + Brainstorming and Evaluating (TEALS Discourse Account Required)

formatted by Markdeep 1.093 

# TEALS Program

[Home](#) | [Curriculum Map](#) | [Additional Readings](#) | [Change Log](#)

## **1** Do Now

---

### **1.1** In your Notebook

---

#### **1.1.1** Write the following

**Pitch** - describe the basic functionality of the app in one paragraph or less

**Define** - List the features/scenarios the app will support

**Sketch** - Draw a very basic wireframe sketch of the main “screens” of the app

**Expand** - Build a comprehensive spec document, leaving out steps or requirements will make it difficult to plan effectively and will likely force major changes or cuts later.

**Plan** - Based on the feature list and spec, create a full development plan making sure to keep an eye on the total amount of time required and to include buffer for things going wrong. Be sure to prioritize tasks so that cuts can be made if necessary.

**Start Coding** - once you have your plan written begin to implement the project according to your plan.

*formatted by [Markdeep 1.093](#)* 

# TEALS Program

[Home](#) | [Curriculum Map](#) | [Additional Readings](#) | [Change Log](#)

## 1 Lesson 8.2 - Defining Requirements

---

### 1.1 Learning Objectives

---

Students will be able to...

- Define key scenarios for a project and the features required to implement each scenario
- Explain the importance of flow charting when designing an application

### 1.2 Materials/Preparation

---

- [Final Project Plan Organizer] handout

### 1.3 Pacing Guide

---

Duration	Description
5 minutes	Do Now
10 minutes	Review pitches
20 minutes	Defining Scenarios
15 minutes	Flow Chart
5 minutes	Debrief and wrap-up

### 1.4 Instructor's Notes

---

#### 1.4.1 1. Do Now

- Project the Do Now on the board, circulate around the class to check that students are working and understand the instructions.

#### 1.4.2 2. Review Pitches

- If desired, give students a few minutes to rework their pitches or get more feedback from a classmate or instructor.
- Ask students to choose which idea they want to pursue, and write it down on the top of their [Final Project Plan Organizer].

#### **1.4.3 3. Scenarios**

- A description of a set of interactions and/or tasks that describe a start-to-finish example of how a user might want to use the application
- Explain that defining scenarios helps a programmer focus on what features are actually necessary to enable the key user interactions for their application
- Instruct students to write down at least **3** scenarios for their project describing, from start to finish, interactions a user might have with their program to accomplish a specific goal
- The scenarios should have a moderate level of detail in the description of the user interaction (e.g. “print a board,” “input their name,” etc.) but should not include any design or implementation details.
- Since this semester is text based a scenario would more likely be: text-based user flows (interactions with the console)
- Once students have written their scenarios, they should review them and develop a list of the necessary features to enable each scenario
- there should be minimal technical detail in these descriptions, instead focusing on details of the user experience. The feature lists should be more about *requirements* than implementation.

#### **1.4.4 3. Flow Chart**

- The different paths a user can take through a program.
- At each step write down what options the user has and what scenarios that can lead to.
- Similar to the wire frame of last semester (wireframes typically deal with screens, so a flow chart is a good alternative)
- Flow Charts do not include any details (such as specific text), but instead provide a broad impression of what an application will be like to aid in design and planning
- Students will complete page 1 of the organizer by writing out the important interactions of their program.
- Encourage students to reference their feature lists to ensure they include *all* necessary interactions for their project, including simple things like a intro interaction, help interaction, or exit (“game over”).

#### **1.4.5 4. Debrief**

- As class ends, ensure students retain their work as they will use it to construct a detailed specification and implementation plan tomorrow.

### **1.5 Accommodation/Differentiation**

---

### **1.6 Forum discussion**

---

Lesson 8.2 - Defining Requirements (TEALS Discourse Account Required)

# TEALS Program

[Home](#) | [Curriculum Map](#) | [Additional Readings](#) | [Change Log](#)

## **1** Do Now

---

### **1.1 Take out your pitches from last class**

---

Review the ideas and pick **one** to pursue for your final project.

*formatted by [Markdeep 1.093](#)* 

# TEALS Program

[Home](#) | [Curriculum Map](#) | [Additional Readings](#) | [Change Log](#)

## 1 Lesson 8.3 - Building a Plan

---

### 1.1 Learning Objectives

---

Students will be able to...

- Identify the main components of a functional project specification and explain the purpose of each section
- Develop a project idea into a full, detailed specification

### 1.2 Materials/Preparation

---

### 1.3 Pacing Guide

---

Duration	Description
5 minutes	Do Now
10 minutes	Review feature lists and flow chart
20 minutes	Spec writing
15 minutes	Building implementation plan
5 minutes	Debrief and wrap-up

### 1.4 Instructor's Notes

---

#### 1.4.1 1. Do Now

- Project the Do Now on the board, circulate around the class to check that students are working and understand the instructions.

#### 1.4.2 2. Review feature lists and flow chart

- If time allows, ask one or two students to share their feature list and/or flow chart and discuss with the class.
- Ensure that students have an understanding of the proper level of detail at this point.

### **1.4.2.1 Spec writing**

- Using the details from their pitch, their feature lists, their flow charts, and the feedback they've received, students should fill out the rest of the [Final Project Plan Organizer].

#### **1.4.2.1.1 Vital**

- At this stage students be as detailed and thorough as they can.
- Any missing information will complicate the process later when they realize what was left out.
- Encourage students to take their time and make sure they hit everything.
- While this process is happening, instructors should circulate through the class and check-in with student.
- Verify that they have a complete, well-thought out idea that is feasible to complete in the available time.
- If you have concerns about a student's ability to complete the proposed project, help them scope down by removing or simplifying features.

### **1.4.3 3. Implementation plan**

- Students should use the details built in their plan organizer to list the tasks necessary on their [Final Project Development Plan].
- Emphasize to students that tasks should be at a very low level of granularity (hence the time requirement being specified in minutes).
- If a single task has a time estimate of more than a few hours, the student should try to break the task into smaller pieces.
- Ensure that students do not skip "trivial" or "simple" tasks (such as building a script they have written before) or non-coding tasks (such as developing graphics) in their plan.

### **1.4.4 4. Debrief**

- As class ends, remind students that their spec and implementation plan will be their guides throughout the process.
- They should update them each day and keep them with them at all times.
- Ideally, anytime there is a question about the requirements or scope of the project, the spec should have the answer.
- If not, it's a new idea and the spec needs to be updated accordingly.

## **1.5 Accommodation/Differentiation**

---

## **1.6 Forum discussion**

---

Lesson 8.3 - Building a Plan (TEALS Discourse Account Required)





# TEALS Program

[Home](#) | [Curriculum Map](#) | [Additional Readings](#) | [Change Log](#)

## 1 Lesson 8.4 - Project Implementation

---

### 1.1 Learning Objectives

---

Students will be able to...

- Use the skills developed throughout the course to implement a medium- to large-scale software project
- Realistically evaluate progress during software development and identify when cuts are necessary
- Prioritize features and scenarios and choose which should be eliminated or modified if/when resources and/or time become limited

### 1.2 Materials/Preparation

---

- Students should each have their [Final Project Plan Organizer](#) and [Final Project Development Plan](#)
- Review [4 Steps to Solve Any CS Problem](#)

### 1.3 Pacing Guide

---

Duration	Description
<i>Days 1-15</i>	
5 minutes	Do Now
10 minutes	Check-in
30 minutes	Lab time
10 minutes	Exit ticket

### 1.4 Instructor Notes

---

#### 1.4.1 1. Do Now

- Project the Do Now on the board, circulate around the class to check that students are working and understand the instructions.

#### **1.4.2 2. Check-in**

- Point out how many days remain and have students check their implementation plan to ensure they do not have more work than time remaining.
- If they do, they will need to create a tentative cut list in case they don't catch up.
- Using previous days exit tickets, questions from students, instructor awareness of trouble points in the project, and/or any other resources to determine what needs covering
- Use this time as an opportunity to remind students of previous labs or activities that may be applicable to their project, and/or how far along they should be by the end of the day

#### **1.4.3 3. Lab time**

- Allow students to work on their project at their own pace
- Provide a mechanism for students to ask questions of course staff as needed
- Simply having students raise hands often does not work well, as it can be hard to keep track of in what order hands were raised; consider a queue of some kind where students write their names when they have a question
- When there are no current questions, circulate and observe progress, stepping in if students appear stuck but are not asking for help
- Be sure to meerkat and not spend more than a minute or two with any single student at a time

#### **1.4.4 3. Exit ticket**

Before students leave, have them answer the following questions on a small piece of paper:

1. What was the last thing you accomplished on the project today?
2. What is the first thing you will work on tomorrow?
3. Are you currently ahead, behind, or on track with your schedule?
4. If you are behind, what tasks will you cut to get back on track?
  - If you are ahead, what are some extra features you can add?
  - What is the riskiest remaining task for your project?
5. These answers will help you determine which students to visit first the next day.
6. Any student who indicates they are behind should get a consult with an instructor the next day to help get them back on track.
7. Encourage students to save each day's version of their planning documents with a new name (possibly using the suffix \_mmdd) so they can track progress and recover cut tasks if they make up time.

### **1.5 Accommodation/Differentiation**

---

### **1.6 Forum discussion**

---

[Lesson 8.4 - Project Implementation \(TEALS Discourse Account Required\)](#)





# TEALS Program

[Home](#) | [Curriculum Map](#) | [Additional Readings](#) | [Change Log](#)

## **1 Do Now Supplemental**

### **1.1 In your Notebook**

#### **1.1.1 Answer the following questions**

1. How could you represent 10 numbers using your fingers?
2. How could you represent the number 1023 using your fingers?

*formatted by [Markdeep 1.093](#)* 

# TEALS Program

[Home](#) | [Curriculum Map](#) | [Additional Readings](#) | [Change Log](#)

## 1 Associated Readings

---

Adapted from **Think Python - How To Think Like A Computer Scientist** by Allen Downey ([HTML Version](#)) ([PDF Version](#))

### 1.1 Unit 1 - Introduction to Python

---

#### 1.2 1.1

---

##### 1.2.1 Values and Types

A value is one of the basic things a program works with, like a letter or a number. The values we have seen so far are 1, 2, and 'Hello, World!'.

These values belong to different types: 2 is an integer, and 'Hello, World!' is a string, so-called because it contains a "string" of letters. You (and the interpreter) can identify strings because they are enclosed in quotation marks.

If you are not sure what type a value has, the interpreter can tell you.

```
>>> type('Hello, World!')
<type 'str'=>

>>> type(17)
<type 'int'=>
```

Not surprisingly, strings belong to the type `str` and integers belong to the type `int`. Less obviously, numbers with a decimal point belong to a type called `float`, because these numbers are represented in a format called floating-point.

```
>>> type(3.2)
<type 'float'=>
```

What about values like '17' and '3.2'? They look like numbers, but they are in quotation marks like strings.

```
>>> type('17')
<type 'str'=>
>>> type('3.2')
<type 'str'=>
```

They're strings. When you type a large integer, you might be tempted to use commas between groups of three digits, as in 1,000,000. This is not a legal integer in Python, but it is legal:

```
>>> 1,000,000
(1, 0, 0)
```

Well, that's not what we expected at all! Python interprets 1,000,000 as a comma-separated sequence of integers. This is the first example we have seen of a semantic error: the code runs without producing an error message, but it doesn't do the "right" thing.

## 1.2.2 Operators and Operands

Operators are special symbols that represent computations like addition and multiplication. The values the operator is applied to are called operands.

The operators `+`, `-`, `*`, `/`, `//` and `**` perform addition, subtraction, multiplication, true division, floor division, and exponentiation, as in the following examples:

```
20+32 hour-1 hour*60+minute minute/60 minute//60 5**2 (5+9)*(15-7)
```

//

Python 3 has two division operators. The `/` operator, also known as *true division*, will always produce a floating point answer. The `//` operator, also known as *floor division*, will round the quotient down to the nearest integer:

```
minute=170
>>> minute / 60
2.8333333333333335
>>> minute // 60
2
```

//

## 1.2.3 Expressions and Statements

An expression is a combination of values, variables, and operators. A value all by itself is considered an expression, and so is a variable, so the following are all legal expressions (assuming that the variable `x` has been assigned a value):

```
17
x
x + 17
```

//

A statement is a unit of code that the Python interpreter can execute. We have seen two kinds of statement: `print` and assignment. Technically an expression is also a statement, but it is probably simpler to think of them as different things. The important difference is that an expression has a value; a statement does not.

## 1.3 1.2

---

### 1.3.1 Variables

One of the most powerful features of a programming language is the ability to manipulate variables. A variable is a name that refers to a value.

An assignment statement creates new variables and gives them values:

```
>>> message = 'And now for something completely different'
>>> n = 17
>>> pi = 3.1415926535897932
```

//

This example makes three assignments. The first assigns a string to a new variable named `message`; the second gives the integer 17 to `n`; the third assigns the (approximate) value of `n` to `pi`. A common way to represent variables on paper is to write the name with an arrow pointing to the variable's value. This kind of figure is called a state diagram because it shows what state each of the variables is in (think of it as the variable's state of mind). Figure 2.1 shows the result of the previous example.

message → 'And now for something completely different'
n → 17
pi → 3.1415926535897932

State diagram

The type of a variable is the type of the value it refers to.

```
>>> type(message)
<type 'str'=>
>>> type(n)
<type 'int'=>
>>> type(pi)
<type 'float'=>
```



### 1.3.2 Variable names and keywords

Programmers generally choose names for their variables that are meaningful—they document what the variable is used for.

Variable names can be arbitrarily long. They can contain both letters and numbers, but they have to begin with a letter. It is legal to use uppercase letters, but it is a good idea to begin variable names with a lowercase letter (you'll see why later).

The underscore character, `_`, can appear in a name. It is often used in names with multiple words, such as `my_name` or `airspeed_of_unladen_swallow`.

If you give a variable an illegal name, you get a syntax error:

```
>>> 76trombones = 'big parade'
SyntaxError: invalid syntax
>>> more@ = 1000000
SyntaxError: invalid syntax
>>> class = 'Advanced Theoretical Zymurgy'
SyntaxError: invalid syntax
```



`76trombones` is illegal because it does not begin with a letter. `more@` is illegal because it contains an illegal character, `@`. But what's wrong with `class`?

It turns out that `class` is one of Python's keywords. The interpreter uses keywords to recognize the structure of the program, and they cannot be used as variable names.

Python 2 has 31 keywords:

```
and del from not while
as elif global or with
assert else if pass yield
break except import print
class exec in raise
continue finally is return
def for lambda try
```



In Python 3, `exec` is no longer a keyword, but `nonlocal` is. You might want to keep this list handy. If the interpreter complains about one of your variable names and you don't know why, see if it is on this list.

## 1.4 1.3

---

### 1.4.1 Debugging

Programming is error-prone. For whimsical reasons, programming errors are called bugs and the process of tracking them down is called debugging.

Three kinds of errors can occur in a program: syntax errors, runtime errors, and semantic errors. It is useful to distinguish between them in order to track them down more quickly.

#### 1.4.1.1 Syntax errors

Python can only execute a program if the syntax is correct; otherwise, the interpreter displays an error message. Syntax refers to the structure of a program and the rules about that structure. For example, parentheses have to come in matching pairs, so `(1 + 2)` is legal, but `8)` is a syntax error.

In English, readers can tolerate most syntax errors, which is why we can read the poetry of e. e. cummings without spewing error messages. Python is not so forgiving. If there is a single syntax error anywhere in your program, Python will display an error message and quit, and you will not be able to run your program. During the first few weeks of your programming career, you will probably spend a lot of time tracking down syntax errors. As you gain experience, you will make fewer errors and find them faster.

#### 1.4.1.2 Runtime errors

The second type of error is a runtime error, so called because the error does not appear until after the program has started running. These errors are also called exceptions because they usually indicate that something exceptional (and bad) has happened.

Runtime errors are rare in the simple programs you will see in the first few chapters, so it might be a while before you encounter one.

#### 1.4.1.3 Semantic errors

The third type of error is the semantic error. If there is a semantic error in your program, it will run successfully in the sense that the computer will not generate any error messages, but it will not do the right thing. It will do something else. Specifically, it will do what you told it to do.

The problem is that the program you wrote is not the program you wanted to write. The meaning of the program (its semantics) is wrong. Identifying semantic errors can be tricky because it requires you to work backward by looking at the output of the program and trying to figure out what it is doing.

#### 1.4.1.4 Experimental debugging

One of the most important skills you will acquire is debugging. Although it can be frustrating, debugging is one of the most intellectually rich, challenging, and interesting parts of programming. In some ways, debugging is like detective work. You are confronted with clues, and you have to infer the processes and events that led to the results you see.

Debugging is also like an experimental science. Once you have an idea about what is going wrong, you modify your program and try again. If your hypothesis was correct, then you can predict the result of the modification, and you take a step closer to a working program. If your hypothesis was wrong, you have to come up with a new one. As Sherlock Holmes pointed out, "When you have eliminated the impossible, whatever remains, however improbable, must be the truth." (A. Conan Doyle, *The Sign of Four*)

For some people, programming and debugging are the same thing. That is, programming is the process of gradually debugging a program until it does what you want. The idea is that you should start with a program that does something and make small modifications, debugging them as you go, so that you always have a working program.

For example, Linux is an operating system that contains thousands of lines of code, but it started out as a simple program Linus Torvalds used to explore the Intel 80386 chip. According to Larry Greenfield, "One of Linus's earlier projects was a program that would switch between printing AAAA and BBBB. This later evolved to Linux." (*The Linux Users' Guide Beta Version 1*).

---

## 1.5 Unit 2 - Data Types, Conditionals, Booleans and Lists

---

### 1.6.1 Values and types

A value is one of the basic things a program works with, like a letter or a number. The values we have seen so far are 1, 2, and 'Hello, World!'.

These values belong to different types: 2 is an integer, and 'Hello, World!' is a string, so-called because it contains a "string" of letters. You (and the interpreter) can identify strings because they are enclosed in quotation marks.

If you are not sure what type a value has, the interpreter can tell you.

```
>>> type('Hello, World!')
<type 'str'=>
>>> type(17)
<type 'int'=>
```

Not surprisingly, strings belong to the type str and integers belong to the type int. Less obviously, numbers with a decimal point belong to a type called float, because these numbers are represented in a format called floating-point.

```
>>> type(3.2)
<type 'float'=>
What about values like '17' and '3.2'? They look like numbers,
but they are in quotation marks like strings.
>>> type('17')
<type 'str'=>
>>> type('3.2')
<type 'str'=>
```

They're strings. When you type a large integer, you might be tempted to use commas between groups of three digits, as in 1,000,000. This is not a legal integer in Python, but it is legal:

```
>>> 1,000,000
(1, 0, 0)
```

Well, that's not what we expected at all! Python interprets 1,000,000 as a comma-separated sequence of integers. This is an example we have seen of a semantic error: the code runs without producing an error message, but it doesn't do the "right" thing.

### 1.6.2 Exercises

#### 1.6.2.1 Exercise 1

Assume that we execute the following assignment statements:

```
width = 17
height = 12.0
delimiter = ','
```

For each of the following expressions, write the value of the expression and the type (of the value of the expression):

- width/2
- width/2.0
- height/3
- 1 + 2 \* 5
- delimiter \* 5

Use the Python interpreter to check your answers.

#### 1.6.2.2 Exercise 2

Practice using the Python interpreter as a calculator:

The volume of a sphere with radius  $r$  is  $\frac{4}{3} \pi r^3$ . What is the volume of a sphere with radius 5? (Hint: 392.7 is wrong!)

Suppose the cover price of a book is \$24.95, but bookstores get a 40% discount. Shipping costs \$3 for the first copy and 75 cents for each additional copy. What is the total wholesale cost for 60 copies?

If I leave my house at 6:52 am and run 1 mile at an easy pace (8:15 per mile), then 3 miles at tempo (7:12 per mile) and 1 mile at easy pace again, what time do I get home for breakfast?

## 1.7 2.2

---

### 1.7.1 Expressions and statements

An expression is a combination of values, variables, and operators. A value all by itself is considered an expression, and so is a variable, so the following are all legal expressions (assuming that the variable  $x$  has been assigned a value):

```
17
x
x + 17
```

A statement is a unit of code that the Python interpreter can execute. We have seen two kinds of statement: print and assignment.

Technically an expression is also a statement, but it is probably simpler to think of them as different things. The important difference is that an expression has a value; a statement does not.

### 1.7.2 Boolean expressions

A Boolean expression is an expression that is either true or false. The following examples use the operator `==`, which compares two operands and produces True if they are equal and False otherwise:

```
>>> 5 == 5
True
>>> 5 == 6
False
```

True and False are special values that belong to the type `bool`; they are not strings:

```
>>> type(True)
<type 'bool'=>
>>> type(False)
<type 'bool'=>
```

The `==` operator is one of the relational operators; the others are:

```
x != y # x is not equal to y
x > y # x is greater than y
x < y # x is less than y
x >= y # x is greater than or equal to y
x <= y # x is less than or equal to y
```

Although these operations are probably familiar to you, the Python symbols are different from the mathematical symbols. A common error is to use a single equal sign (`=`) instead of a double equal sign (`==`). Remember that `=` is an assignment operator and `==` is a relational operator. There is no such thing as `=<` or `=>`.

### 1.7.3 Logical operators

There are three logical operators: `and`, `or`, and `not`. The semantics (meaning) of these operators is similar to their meaning in English. For example,  $x > 0$  and  $x < 10$  is true only if  $x$  is greater than 0 and less than 10.

`n * 2 == 4 or n * 10 == 100` is true if either of the conditions is true, that is, if the number is 2 or 10.

The `or` expression is true whenever one of the operands is true and the `and` expression is false whenever one of the operands is false.

Finally, the `not` operator negates a Boolean expression, so `not (x > y)` is true if  $x > y$  is false, that is, if  $x$  is less than or equal to  $y$ .

Strictly speaking, the operands of the logical operators should be Boolean expressions, but Python is not very strict. Any nonzero number is interpreted as “true.”

```
>>> 17 and True
True
```

This flexibility can be useful, but there are some subtleties to it that might be confusing. You might want to avoid it (unless you know what you are doing).

## 1.8 2.3

---

### 1.8.1 Conditional execution

In order to write useful programs, we almost always need the ability to check conditions and change the behavior of the program accordingly. Conditional statements give us this ability. The simplest form is the `if` statement:

```
if x > 0:
 print('x is positive')
```

The Boolean expression after `if` is called the condition. If it is true, then the indented statement gets executed. If not, nothing happens.

`if`-statements have the same structure as function definitions: a header followed by an indented body. Statements like this are called compound statements.

There is no limit on the number of statements that can appear in the body, but there has to be at least one. Occasionally, it is useful to have a body with no statements (usually as a place keeper for code you haven’t written yet). In that case, you can use the `pass` statement, which does nothing.

```
if x < 0:
 pass # need to handle negative values!
Alternative execution
```

A second form of the `if` statement is alternative execution, in which there are two possibilities and the condition determines which one gets executed. The syntax looks like this:

```
if x%2 == 0:
 print('x is even')
else:
 print('x is odd')
```

If the remainder when  $x$  is divided by 2 is 0, then we know that  $x$  is even, and the program displays a message to that effect. If the condition is false, the second set of statements is executed. Since the condition must be true or false, exactly one of the alternatives will be executed. The alternatives are called branches, because they are branches in the flow of execution.

### 1.8.2 Chained conditionals

Sometimes there are more than two possibilities and we need more than two branches. One way to express a computation like that is a chained conditional:

```
if x < y:
 print('x is less than y')
elif x > y:
 print('x is greater than y')
else:
 print('x and y are equal')
```



elif is an abbreviation of “else if.” Again, exactly one branch will be executed. There is no limit on the number of elif statements. If there is an else clause, it has to be at the end, but there doesn’t have to be one.

```
if choice == 'a':
 draw_a()
elif choice == 'b':
 draw_b()
elif choice == 'c':
 draw_c()
```



Each condition is checked in order. If the first is false, the next is checked, and so on. If one of them is true, the corresponding branch executes, and the statement ends. Even if more than one condition is true, only the first true branch executes.

### 1.8.3 Nested conditionals

One conditional can also be nested within another. We could have written the trichotomy example like this:

```
if x == y:
 print('x and y are equal')
else:
 if x < y:
 print('x is less than y')
 else:
 print('x is greater than y')
```



The outer conditional contains two branches. The first branch contains a simple statement. The second branch contains another if statement, which has two branches of its own. Those two branches are both simple statements, although they could have been conditional statements as well.

Although the indentation of the statements makes the structure apparent, nested conditionals become difficult to read very quickly. In general, it is a good idea to avoid them when you can.

Logical operators often provide a way to simplify nested conditional statements. For example, we can rewrite the following code using a single conditional:

```
if 0 < x:
 if x < 10:
 print('x is a positive single-digit number.')
```



The print(statement) is executed only if we make it past both conditionals, so we can get the same effect with the and operator:)

```
if 0 < x and x < 10:
 print('x is a positive single-digit number.')
```



## 1.9 2.4

---

### 1.9.1 A list is a sequence

Like a string, a list is a sequence of values. In a string, the values are characters; in a list, they can be any type. The values in a list are called elements or sometimes items.

There are several ways to create a new list; the simplest is to enclose the elements in square brackets ([ and ]):

```
[10, 20, 30, 40]
['crunchy frog', 'ram bladder', 'lark vomit']
```

The first example is a list of four integers. The second is a list of three strings. The elements of a list don't have to be the same type. The following list contains a string, a float, an integer, and (lo!) another list:

```
['spam', 2.0, 5, [10, 20]]
```

A list within another list is nested.

A list that contains no elements is called an empty list; you can create one with empty brackets, [].

As you might expect, you can assign list values to variables:

```
>>> cheeses = ['Cheddar', 'Edam', 'Gouda']
>>> numbers = [17, 123]
>>> empty = []
>>> print(cheeses, numbers, empty)
['Cheddar', 'Edam', 'Gouda'] [17, 123] []
```

## 1.9.2 Lists are mutable

The syntax for accessing the elements of a list is the same as for accessing the characters of a string—the bracket operator. The expression inside the brackets specifies the index. Remember that the indices start at 0:

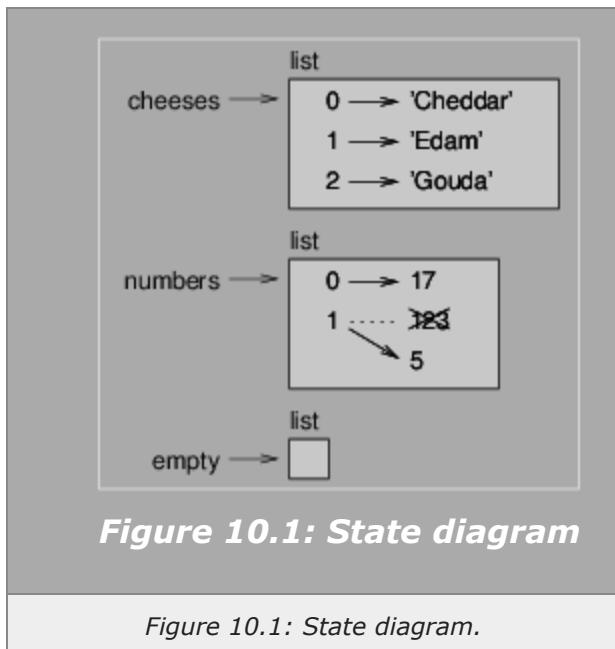
```
>>> print(cheeses[0])
Cheddar
```

Unlike strings, lists are mutable. When the bracket operator appears on the left side of an assignment, it identifies the element of the list that will be assigned.

```
>>> numbers = [17, 123]
>>> numbers[1] = 5
>>> print(numbers)
[17, 5]
```

The one-eth element of numbers, which used to be 123, is now 5.

You can think of a list as a relationship between indices and elements. This relationship is called a mapping; each index “maps to” one of the elements. Figure 10.1 shows the state diagram for cheeses, numbers and empty:



Lists are represented by boxes with the word “list” outside and the elements of the list inside. cheeses refers to a list with three elements indexed 0, 1 and 2. numbers contains two elements; the diagram shows that the value of the second element has been reassigned from 123 to 5. empty refers to a list with no elements.

List indices work the same way as string indices:

Any integer expression can be used as an index. If you try to read or write an element that does not exist, you get an IndexError. If an index has a negative value, it counts backward from the end of the list.

## 1.10 2.5

---

### 1.10.1 List operations

The + operator concatenates lists:

```
>>> a = [1, 2, 3]
>>> b = [4, 5, 6]
>>> c = a + b
>>> print(c)
[1, 2, 3, 4, 5, 6]
```

Similarly, the \* operator repeats a list a given number of times:

```
>>> [0] * 4
[0, 0, 0, 0]
>>> [1, 2, 3] * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

The first example repeats [0] four times. The second example repeats the list [1, 2, 3] three times.

### 1.10.2 List slices

The slice operator works on lists:

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> t[1:3]
['b', 'c']
>>> t[:4]
['a', 'b', 'c', 'd']
>>> t[3:]
['d', 'e', 'f']
```

If you omit the first index, the slice starts at the beginning. If you omit the second, the slice goes to the end. So if you omit both, the slice is a copy of the whole list.

```
>>> t[:]
['a', 'b', 'c', 'd', 'e', 'f']
```

Since lists are mutable, it is often useful to make a copy before performing operations that fold, spindle or mutilate lists.

A slice operator on the left side of an assignment can update multiple elements:

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> t[1:3] = ['x', 'y']
>>> print(t)
['a', 'x', 'y', 'd', 'e', 'f']
```

### 1.10.3 List methods

Python provides methods that operate on lists. For example, append adds a new element to the end of a list:

```
>>> t = ['a', 'b', 'c']
>>> t.append('d')
>>> print(t)
['a', 'b', 'c', 'd'] //
```

#### 1.10.4 Deleting elements

There are several ways to delete elements from a list. If you know the index of the element you want, you can use pop:

```
>>> t = ['a', 'b', 'c']
>>> x = t.pop(1)
>>> print(t)
['a', 'c']
>>> print(x)
b //
```

pop modifies the list and returns the element that was removed. If you don't provide an index, it deletes and returns the last element.

If you know the element you want to remove you can use remove:

```
>>> t = ['a', 'b', 'c']
>>> t.remove('b')
>>> print(t)
['a', 'c'] //
```

### 1.11 2.6

---

The in operator works on lists.

```
>>> cheeses = ['Cheddar', 'Edam', 'Gouda']
>>> 'Edam' in cheeses
True
>>> 'Brie' in cheeses
False //
```

### 1.12 2.7

---

#### 1.12.1 The while statement

Computers are often used to automate repetitive tasks. Repeating identical or similar tasks without making errors is something that computers do well and people do poorly.

Here is an example countdown snippet that uses a while statement:

```
while n > 0:
 print(n)
 n = n-1
print('Blastoff!') //
```

You can almost read the while statement as if it were English. It means, "While n is greater than 0, display the value of n and then reduce the value of n by 1. When you get to 0, display the word Blastoff!"

More formally, here is the flow of execution for a while statement:

1. Evaluate the condition, yielding True or False.
2. If the condition is false, exit the while statement and continue execution at the next statement.
3. If the condition is true, execute the body and then go back to step 1.

This type of flow is called a loop because the third step loops back around to the top.

The body of the loop should change the value of one or more variables so that eventually the condition becomes false and the loop terminates. Otherwise the loop will repeat forever, which is called an infinite loop. An endless source of amusement for computer scientists is the observation that the directions on shampoo, "Lather, rinse, repeat," are an infinite loop.

In the case of countdown, we can prove that the loop terminates because we know that the value of  $n$  is finite, and we can see that the value of  $n$  gets smaller each time through the loop, so eventually we have to get to 0. In other cases, it is not so easy to tell:

```
while n != 1:
 print(n)
 if n%2 == 0: # n is even
 n = n//2
 else: # n is odd
 n = n*3+1
```



The condition for this loop is  $n \neq 1$ , so the loop will continue until  $n$  is 1, which makes the condition false.

Each time through the loop, the program outputs the value of  $n$  and then checks whether it is even or odd. If it is even,  $n$  is divided by 2. If it is odd, the value of  $n$  is replaced with  $n*3+1$ . For example, if the argument passed to sequence is 3, the resulting sequence is 3, 10, 5, 16, 8, 4, 2, 1.

Since  $n$  sometimes increases and sometimes decreases, there is no obvious proof that  $n$  will ever reach 1, or that the program terminates. For some particular values of  $n$ , we can prove termination. For example, if the starting value is a power of two, then the value of  $n$  will be even each time through the loop until it reaches 1. The previous example ends with such a sequence, starting with 16.

## 1.13 Unit 3 - Functions

---

### 1.14 3.1

---

#### 1.14.1 Function calls

In the context of programming, a function is a named sequence of statements that performs a computation. When you define a function, you specify the name and the sequence of statements. Later, you can "call" the function by name. We have already seen one example of a function call:

```
>>> type(32)
<type 'int'=>
```



The name of the function is `type`. The expression in parentheses is called the argument of the function. The result, for this function, is the type of the argument.

It is common to say that a function "takes" an argument and "returns" a result. The result is called the return value.

#### 1.14.2 Type conversion functions

Python provides built-in functions that convert values from one type to another. The `int` function takes any value and converts it to an integer, if it can, or complains otherwise:

```
>>> int('32')
32
>>> int('Hello')
ValueError: invalid literal for int(): Hello
```



`int` can convert floating-point values to integers, but it doesn't round off; it chops off the fraction part:

```
>>> int(3.99999)
3
>>> int(-2.3)
-2
float converts integers and strings to floating-point numbers:
>>> float(32)
32.0
>>> float('3.14159')
3.14159
Finally, str converts its argument to a string:
>>> str(32)
'32'
>>> str(3.14159)
'3.14159'
```

### 1.14.3 Math functions

Python has a math module that provides most of the familiar mathematical functions. A module is a file that contains a collection of related functions.

Before we can use the module, we have to import it:

```
>>> import math
This statement creates a module object named math.
If you print the module object, you get some information about it:
>>> print(math)
<module 'math'="" (built-in)="">
```

The module object contains the functions and variables defined in the module. To access one of the functions, you have to specify the name of the module and the name of the function, separated by a dot (also known as a period). This format is called dot notation.

```
>>> ratio = signal_power / noise_power
>>> decibels = 10 * math.log10(ratio)

>>> radians = 0.7
>>> height = math.sin(radians)
```

The first example uses log10 to compute a signal-to-noise ratio in decibels (assuming that `signal_power` and `noise_power` are defined). The math module also provides log, which computes logarithms base e.

The second example finds the sine of radians. The name of the variable is a hint that sin and the other trigonometric functions (cos, tan, etc.) take arguments in radians. To convert from degrees to radians, divide by 360 and multiply by  $2\pi$ :

```
>>> degrees = 45
>>> radians = degrees / 360.0 * 2 * math.pi
>>> math.sin(radians)
0.707106781187
```

The expression `math.pi` gets the variable pi from the math module. The value of this variable is an approximation of  $\pi$ , accurate to about 15 digits.

If you know your trigonometry, you can check the previous result by comparing it to the square root of two divided by two:

```
>>> math.sqrt(2) / 2.0
0.707106781187
```

### 1.14.4 Composition

So far, we have looked at the elements of a program—variables, expressions, and statements—in isolation, without talking about how to combine them.

One of the most useful features of programming languages is their ability to take small building blocks and compose them. For example, the argument of a function can be any kind of expression, including arithmetic operators:

```
x = math.sin(degrees / 360.0 * 2 * math.pi)
And even function calls:
x = math.exp(math.log(x+1))
```

Almost anywhere you can put a value, you can put an arbitrary expression, with one exception: the left side of an assignment statement has to be a variable name. Any other expression on the left side is a syntax error (we will see exceptions to this rule later).

```
>>> minutes = hours * 60 # right
>>> hours * 60 = minutes # wrong!
SyntaxError: can't assign to operator
```

## 1.15 3.2

### 1.15.1 Adding new functions

So far, we have only been using the functions that come with Python, but it is also possible to add new functions. A function definition specifies the name of a new function and the sequence of statements that execute when the function is called.

Here is an example:

```
def print_lyrics():
 print("I'm a lumberjack, and I'm okay.")
 print("I sleep all night and I work all day.")
```

`def` is a keyword that indicates that this is a function definition. The name of the function is `print_lyrics`. The rules for function names are the same as for variable names: letters, numbers and some punctuation marks are legal, but the first character can't be a number. You can't use a keyword as the name of a function, and you should avoid having a variable and a function with the same name.

The empty parentheses after the name indicate that this function doesn't take any arguments.

The first line of the function definition is called the header; the rest is called the body. The header has to end with a colon and the body has to be indented. By convention, the indentation is always four spaces. The body can contain any number of statements.

The strings in the `print` statements are enclosed in double quotes. Single quotes and double quotes do the same thing; most people use single quotes except in cases like this where a single quote (which is also an apostrophe) appears in the string.

If you type a function definition in interactive mode, the interpreter prints ellipses to let you know that the definition isn't complete:

```
>>> def print_lyrics():
... print("I'm a lumberjack, and I'm okay.")
... print("I sleep all night and I work all day.")
...
```

To end the function, you have to enter an empty line (this is not necessary in a script). Defining a function creates a variable with the same name.

```
>>> print(print_lyrics)
<function print_lyrics=' at="" 0xb7e99e9c="">
>>> type(print_lyrics)
<type 'function'='>
```

The value of `print_lyrics` is a function object, which has type 'function'. The syntax for calling the new function is the same as for built-in functions:

```
>>> print_lyrics()
I'm a lumberjack, and I'm okay.
I sleep all night and I work all day.
```

Once you have defined a function, you can use it inside another function. For example, to repeat the previous refrain, we could write a function called `repeat_lyrics`:

```
def repeat_lyrics():
 print_lyrics()
 print_lyrics()
```

And then call `repeat_lyrics`:

```
>>> repeat_lyrics()
I'm a lumberjack, and I'm okay.
I sleep all night and I work all day.
I'm a lumberjack, and I'm okay.
I sleep all night and I work all day.
But that's not really how the song goes.
```

### 1.15.2 Definitions and uses

Pulling together the code fragments from the previous section, the whole program looks like this:

```
def print_lyrics():
 print("I'm a lumberjack, and I'm okay.")
 print("I sleep all night and I work all day.")

def repeat_lyrics():
 print_lyrics()
 print_lyrics()

repeat_lyrics()
```

This program contains two function definitions: `print_lyrics` and `repeat_lyrics`. Function definitions get executed just like other statements, but the effect is to create function objects. The statements inside the function do not get executed until the function is called, and the function definition generates no output.

As you might expect, you have to create a function before you can execute it. In other words, the function definition has to be executed before the first time it is called.

### 1.15.3 Exercise 3.1

Move the last line of this program to the top, so the function call appears before the definitions. Run the program and see what error message you get.

### 1.15.4 Exercise 3.2

Move the function call back to the bottom and move the definition of `print_lyrics` after the definition of `repeat_lyrics`. What happens when you run this program?

### 1.15.5 Flow of execution

In order to ensure that a function is defined before its first use, you have to know the order in which statements are executed, which is called the flow of execution.

Execution always begins at the first statement of the program. Statements are executed one at a time, in order from top to bottom.

Function definitions do not alter the flow of execution of the program, but remember that statements inside the function are not executed until the function is called.

A function call is like a detour in the flow of execution. Instead of going to the next statement, the flow jumps to the body of the function, executes all the statements there, and then comes back to pick up where it left off.

That sounds simple enough, until you remember that one function can call another. While in the middle of one function, the program might have to execute the statements in another function. But while executing that new function, the program might have to execute yet another function!

Fortunately, Python is good at keeping track of where it is, so each time a function completes, the program picks up where it left off in the function that called it. When it gets to the end of the program, it terminates.

What's the moral of this sordid tale? When you read a program, you don't always want to read from top to bottom. Sometimes it makes more sense if you follow the flow of execution.

### 1.15.6 Parameters and arguments

Some of the built-in functions we have seen require arguments. For example, when you call `math.sin` you pass a number as an argument. Some functions take more than one argument: `math.pow` takes two, the base and the exponent.

Inside the function, the arguments are assigned to variables called parameters. Here is an example of a user-defined function that takes an argument:

```
def print_twice(bruce):
 print(bruce)
 print(bruce)
```

This function assigns the argument to a parameter named `Bruce`. When the function is called, it prints the value of the parameter (whatever it is) twice.

This function works with any value that can be printed.

```
>>> print_twice('Spam')
Spam
Spam
>>> print_twice(17)
17
17
>>> print_twice(math.pi)
3.14159265359
3.14159265359
```

The same rules of composition that apply to built-in functions also apply to user-defined functions, so we can use any kind of expression as an argument for `print_twice`:

```
>>> print_twice('Spam '*4)
Spam Spam Spam Spam
Spam Spam Spam Spam
>>> print_twice(math.cos(math.pi))
-1.0
-1.0
```

The argument is evaluated before the function is called, so in the examples the expressions `'Spam '*4` and `math.cos(math.pi)` are only evaluated once.

You can also use a variable as an argument:

```
>>> michael = 'Eric, the half a bee.'
>>> print_twice(michael)
Eric, the half a bee.
Eric, the half a bee.
```

The name of the variable we pass as an argument (`Michael`) has nothing to do with the name of the parameter (`Bruce`). It doesn't matter what the value was called back home (in the caller); here in `print_twice`, we call everybody `Bruce`.

## 1.16 3.3

### 1.16.1 Return values

Some of the built-in functions we have used, such as the math functions, produce results. Calling the function generates a value, which we usually assign to a variable or use as part of an expression.

```
e = math.exp(1.0)
height = radius * math.sin(radians)
```

All of the functions we have written so far are void; they print something or move turtles around, but their return value is None.

In this chapter, we are (finally) going to write fruitful functions. The first example is area, which returns the area of a circle with the given radius:

```
def area(radius):
 temp = math.pi * radius**2
 return temp
```

This statement means: "Return immediately from this function and use the following expression as a return value." The expression can be arbitrarily complicated, so we could have written this function more concisely:

```
def area(radius):
 return math.pi * radius**2
```

On the other hand, temporary variables like temp often make debugging easier.

Sometimes it is useful to have multiple return statements, one in each branch of a conditional:

```
def absolute_value(x):
 if x < 0:
 return -x
 else:
 return x
```

Since these return statements are in an alternative conditional, only one will be executed.

As soon as a return statement executes, the function terminates without executing any subsequent statements. Code that appears after a return statement, or any other place the flow of execution can never reach, is called dead code.

It is a good idea to ensure that every possible path through the program hits a return statement. For example:

```
def absolute_value(x):
 if x < 0:
 return -x
 if x > 0:
 return x
```

This function is incorrect because if x happens to be 0, neither condition is true, and the function ends without hitting a return statement. If the flow of execution gets to the end of a function, the return value is None, which is not the absolute value of 0.

```
>>> print(absolute_value(0))
None
```

By the way, Python provides a built-in function called abs that computes absolute values.

#### 1.16.1.1 Exercise 3.3

Write a compare function that returns 1 if  $x > y$ , 0 if  $x == y$ , and  $-1$  if  $x < y$ .

### 1.17 3.4

### 1.17.1 Aliasing

If a refers to an object and you assign b = a, then both variables refer to the same object:

```
>>> a = [1, 2, 3]
>>> b = a
>>> b is a
True
```

The association of a variable with an object is called a reference. In this example, there are two references to the same object.

An object with more than one reference has more than one name, so we say that the object is aliased.

If the aliased object is mutable, changes made with one alias affect the other:

```
>>> b[0] = 17
>>> print(a)
[17, 2, 3]
```

Although this behavior can be useful, it is error-prone. In general, it is safer to avoid aliasing when you are working with mutable objects.

### 1.17.2 Variables and parameters are local

When you create a variable inside a function, it is local, which means that it only exists inside the function. For example:

```
def cat_twice(part1, part2):
 cat = part1 + part2
 print_twice(cat)
```

This function takes two arguments, concatenates them, and prints the result twice. Here is an example that uses it:

```
>>> line1 = 'Bing tiddle '
>>> line2 = 'tiddle bang.'
>>> cat_twice(line1, line2)
Bing tiddle tiddle bang.
Bing tiddle tiddle bang.
```

When cat\_twice terminates, the variable cat is destroyed. If we try to print it, we get an exception:

```
>>> print(cat)
NameError: name 'cat' is not defined
```

Parameters are also local. For example, outside print\_twice, there is no such thing as Bruce.

### 1.17.3 Global variables

Variables created outside the function, so it belongs to the special frame called `__main__`. Variables in `__main__` are sometimes called global because they can be accessed from any function. Unlike local variables, which disappear when their function ends, global variables persist from one function call to the next.

It is common to use global variables for constants; that is, variables that do not change. For example, some programs use constants to indicate the minimum or maximum number of a dataset like the max level of a game could be set to 10.

If you try to reassign a global variable, you might be surprised. The following example is supposed to keep track of whether the function has been called:

```
been_called = False
```

```
def example2():
 been_called = True # WRONG
```

But if you run it you will see that the value of `been_called` doesn't change. The problem is that `example2` creates a new local variable named `been_called`. The local variable goes away when the function ends, and has no effect on the global variable.

To reassign a global variable inside a function you have to declare the global variable before you use it:

```
been_called = False

def example2():
 global been_called
 been_called = True
```

The `global` statement tells the interpreter something like, "In this function, when I say `been_called`, I mean the global variable; don't create a local one."

Here's an example that tries to update a global variable:

```
count = 0

def example3():
 count = count + 1 # WRONG
```

If you run it you get the following error message:

```
UnboundLocalError: local variable 'count' referenced before assignment
```

Python assumes that `count` is local, and under that assumption you are reading it before writing it. The solution, again, is to declare `count` global.

```
def example3():
 global count
 count += 1
```

If a global variable refers to a mutable value, you can modify the value without declaring the variable global:

```
known = [10, 20]

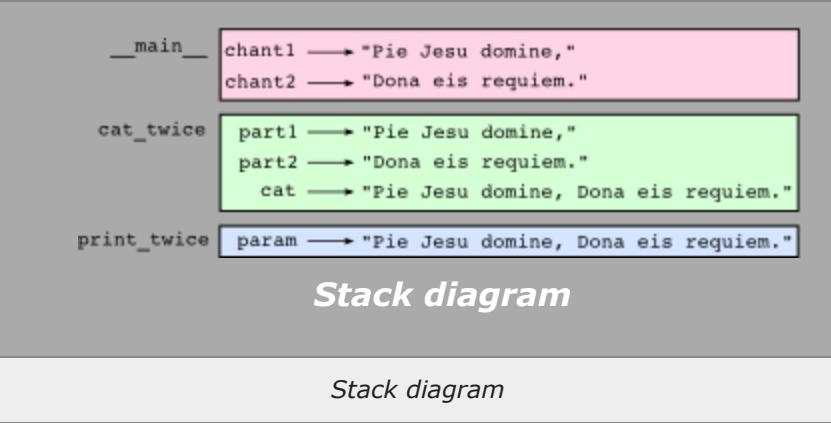
def example4():
 known[1] = 30
```

If a program has a lot of global variables and are modify them frequently, they can make programs hard to debug.

#### 1.17.4 Stack diagrams

To keep track of which variables can be used where, it is sometimes useful to draw a stack diagram. Like state diagrams, stack diagrams show the value of each variable, but they also show the function to which each variable belongs.

Each function is represented by a frame. A frame is a box with the name of a function beside it and the parameters and variables of the function inside it. The stack diagram for the previous example looks like this:



The order of the stack shows the flow of execution. `print_twice` was called by `cat_twice`, and `cat_twice` was called by `__main__`, which is a special name for the topmost function. When you create a variable outside of any function, it belongs to `__main__`.

Each parameter refers to the same value as its corresponding argument. So, `part1` has the same value as `chant1`, `part2` has the same value as `chant2`, and `param` has the same value as `cat`.

If an error occurs during a function call, Python prints the name of the function, and the name of the function that called it, and the name of the function that called that, all the way back to the top most function.

To see how this works, create a Python script named `tryme2.py` that looks like this:

```
def print_twice(param):
 print(param)
 print(param)
 print(cat)

def cat_twice(part1, part2):
 cat = part1 + part2
 print_twice(cat)

chant1 = "Pie Jesu domine, "
chant2 = "Dona eis requim."
cat_twice(chant1, chant2)
```

We've added the statement, `print(cat)` inside the `print_twice` function, but `cat` is not defined there. Running this script will produce an error message like this:

```
Traceback (innermost last):
 File "tryme2.py", line 12, in <module>
 cat_twice(chant1, chant2)
 File "tryme2.py", line 8, in cat_twice
 print_twice(cat)
 File "tryme2.py", line 4, in print_twice
 print(cat)
NameError: global name 'cat' is not defined
```

This list of functions is called a traceback. It tells you what program file the error occurred in, and what line, and what functions were executing at the time. It also shows the line of code that caused the error.

Notice the similarity between the traceback and the stack diagram. It's not a coincidence. In fact, another common name for a traceback is a stack trace.

## 1.17.5 List arguments

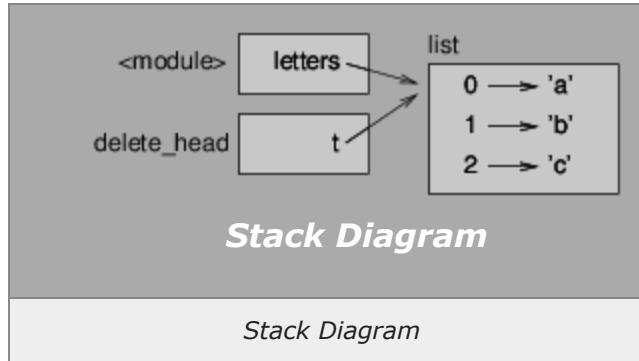
When you pass a list to a function, the function gets a reference to the list. If the function modifies a list parameter, the caller sees the change. For example, `delete_head` removes the first element from a list:

```
def delete_head(t):
 del t[0]
```

Here's how it is used:

```
>>> letters = ['a', 'b', 'c']
>>> delete_head(letters)
>>> print(letters)
['b', 'c']
```

The parameter `t` and the variable `letters` are aliases for the same object. The stack diagram looks like the following:



Since the list is shared by two frames, I drew it between them.

It is important to distinguish between operations that modify lists and operations that create new lists. For example, the `append` method modifies a list, but the `+` operator creates a new list:

```
>>> t1 = [1, 2]
>>> t2 = t1.append(3)
>>> print(t1)
[1, 2, 3]
>>> print(t2)
None

>>> t3 = t1 + [4]
>>> print(t3)
[1, 2, 3, 4]
```

This difference is important when you write functions that are supposed to modify lists. For example, this function does not delete the head of a list:

```
def bad_delete_head(t):
 t = t[1:] # WRONG!
```

The slice operator creates a new list and the assignment makes `t` refer to it, but none of that has any effect on the list that was passed as an argument.

An alternative is to write a function that creates and returns a new list. For example, `tail` returns all but the first element of a list:

```
def tail(t):
 return t[1:]
```

This function leaves the original list unmodified. Here's how it is used:

```
>>> letters = ['a', 'b', 'c']
>>> rest = tail(letters)
>>> print(rest)
['b', 'c']
```

## 1.18 Unit 4 - Nested Loops and Lists

### 1.19 4.1

The most common way to traverse the elements of a list is with a for loop. The syntax is the same as for strings:

```
for cheese in cheeses:
 print(cheese)
```

This works well if you only need to read the elements of the list. A for loop over an empty list never executes the body:

```
for x in []:
 print('This never happens.')
```

Although a list can contain another list, the nested list still counts as a single element. The length of this list is four:

```
['spam', 1, ['Brie', 'Roquefort', 'Pol le Veq'], [1, 2, 3]]
```

### 1.19.1 Traversing a String

A lot of computations involve processing a string one character at a time. Often they start at the beginning, select each character in turn, do something to it, and continue until the end. This pattern of processing is called a traversal. One way to write a traversal is with a while loop:

```
index = 0
while index < len(fruit):
 letter = fruit[index]
 print(letter)
 index = index + 1
```

This loop traverses the string and displays each letter on a line by itself. The loop condition is `index < len(fruit)`, so when `index` is equal to the length of the string, the condition is false, and the body of the loop is not executed. The last character accessed is the one with the index `len(fruit)-1`, which is the last character in the string.

#### 1.19.1.1 Exercise 4.1

Write a function that takes a string as an argument and displays the letters backward, one per line.

Another way to write a traversal is with a for loop:

```
for char in fruit:
 print(char)
```

Each time through the loop, the next character in the string is assigned to the variable `char`. The loop continues until no characters are left.

The following example shows how to use concatenation (string addition) and a for loop to generate an abecedarian series (that is, in alphabetical order). In Robert McCloskey's book *Make Way for Ducklings*, the names of the ducklings are Jack, Kack, Lack, Mack, Nack, Ouack, Pack, and Quack. This loop outputs these names in order:

```
prefixes = 'JKLMNOPQ'
suffix = 'ack'

for letter in prefixes:
 print(letter + suffix)
```

The output is:

```
Jack
Kack
Lack
Mack
Nack
Ouack
```

Pack  
Qack



Of course, that's not quite right because Ouack and Quack are misspelled.

### 1.19.1.2 Exercise 4.2

Modify the program to fix this error.

## 1.19.2 Looping and counting

The following program counts the number of times the letter a appears in a string:

```
word = 'banana'
count = 0
for letter in word:
 if letter == 'a':
 count = count + 1
print(count)
```



This program demonstrates another pattern of computation called a counter. The variable count is initialized to 0 and then incremented each time an a is found. When the loop exits, count contains the result—the total number of a's.

### 1.19.3 Exercise 4.3

Encapsulate this code in a function named count, and generalize it so that it accepts the string and the letter as arguments.

### 1.19.4 Exercise 4.4

Rewrite this function so that instead of traversing the string, it uses the three-parameter version of find from the previous section.

## 1.20 4.2

---

### 1.20.1 For Loop Using Range

As mentioned, the most common way to traverse the elements of a list is with a for loop. If you want to write or update the elements, you need the indices. A common way to do that is to combine the functions range and len:

```
for i in range(len(numbers)):
 numbers[i] = numbers[i] * 2
```



This loop traverses the list and updates each element. len returns the number of elements in the list. range returns a list of indices from 0 to n-1, where n is the length of the list. Each time through the loop i gets the index of the next element. The assignment statement in the body uses i to read the old value of the element and to assign the new value.

### 1.20.2 Debugging 4

When you use indices to traverse the values in a sequence, it is tricky to get the beginning and end of the traversal right. Here is a function that is supposed to compare two words and return True if one of the words is the reverse of the other, but it contains two errors:

```
def is_reverse(word1, word2):
 if len(word1) != len(word2):
 return False

 i = 0
 j = len(word2)

 while j > 0:
 if word1[i] != word2[j]:
 return False
 i = i+1
 j = j-1

 return True
```



The first if statement checks whether the words are the same length. If not, we can return False immediately and then, for the rest of the function, we can assume that the words are the same length. This is an example of the guardian pattern in Section 6.8.

i and j are indices: i traverses word1 forward while j traverses word2 backward. If we find two letters that don't match, we can return False immediately. If we get through the whole loop and all the letters match, we return True.

If we test this function with the words "pots" and "stop", we expect the return value True, but we get an IndexError:

```
>>> is_reverse('pots', 'stop')
...
File "reverse.py", line 15, in is_reverse
 if word1[i] != word2[j]:
IndexError: string index out of range
```



For debugging this kind of error, my first move is to print the values of the indices immediately before the line where the error appears.

```
while j > 0:
 print(i, j) # print here

 if word1[i] != word2[j]:
 return False
 i = i+1
 j = j-1
```



Now when I run the program again, I get more information:

```
>>> is_reverse('pots', 'stop')
0 4
...
IndexError: string index out of range
```

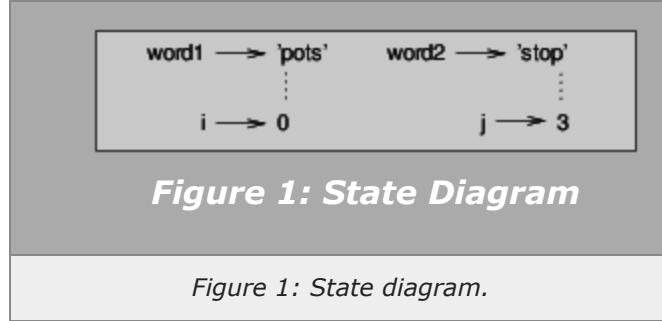


The first time through the loop, the value of j is 4, which is out of range for the string 'pots'. The index of the last character is 3, so the initial value for j should be len(word2)-1. If I fix that error and run the program again, I get:

```
>>> is_reverse('pots', 'stop')
0 3
1 2
2 1
True
```



This time we get the right answer, but it looks like the loop only ran three times, which is suspicious. To get a better idea of what is happening, it is useful to draw a state diagram. During the first iteration, the frame for is\_reverse is shown in Figure 1.



I took a little license by arranging the variables in the frame and adding dotted lines to show that the values of `i` and `j` indicate characters in `word1` and `word2`.

### 1.20.2.1 Exercise 5.1

Starting with this diagram, execute the program on paper, changing the values of `i` and `j` during each iteration. Find and fix the second error in this function.

## 1.21 Unit 6 - Dictionaries

---

### 1.22 6.1

---

A dictionary is like a list, but more general. In a list, the indices have to be integers; in a dictionary they can be (almost) any type.

You can think of a dictionary as a mapping between a set of indices (which are called keys) and a set of values. Each key maps to a value. The association of a key and a value is called a key-value pair or sometimes an item.

As an example, we'll build a dictionary that maps from English to Spanish words, so the keys and the values are all strings.

The function `dict` creates a new dictionary with no items. Because `dict` is the name of a built-in function, you should avoid using it as a variable name.

```
>>> eng2sp = dict()
>>> print(eng2sp)
{}
```



The squiggly-brackets, `{}`, represent an empty dictionary.

You can create a new dictionary with three items:

```
>>> eng2sp = {'one': 'uno', 'two': 'dos', 'three': 'tres'}
```



But if you print `eng2sp`, you might be surprised:

```
>>> print(eng2sp)
{'one': 'uno', 'three': 'tres', 'two': 'dos'}
```



The order of the key-value pairs is not the same. In fact, if you type the same example on your computer, you might get a different result. In general, the order of items in a dictionary is unpredictable.

But that's not a problem because the elements of a dictionary are never indexed with integer indices. Instead, you use the keys to look up the corresponding values:

```
>>> print eng2sp['two']
'dos'
```



The key 'two' always maps to the value 'dos' so the order of the items doesn't matter. If the key isn't in the dictionary, you get an exception:

```
>>> print eng2sp['four']
KeyError: 'four'
```

Dictionaries have a method called `get` that takes a key and a default value. If the key appears in the dictionary, `get` returns the corresponding value; otherwise it returns the default value. For example:

```
>>> h = histogram('a')
>>> print h
{'a': 1}
>>> h.get('a', 0)
1
>>> h.get('b', 0)
0
```

The `len` function works on dictionaries; it returns the number of key-value pairs:

```
>>> len(eng2sp)
3
```

The `in` operator works on dictionaries; it tells you whether something appears as a key in the dictionary (appearing as a value is not good enough).

```
>>> 'one' in eng2sp
True
>>> 'uno' in eng2sp
False
```

## 1.23 6.2

---

To add items to the dictionary, you can use square brackets:

```
>>> eng2sp = {}
>>> print(eng2sp)
{}
>>> eng2sp['one'] = 'uno'
```

This line creates an item that maps from the key `one` to the value `uno`. If we print the dictionary again, we see a key-value pair with a colon between the key and value:

```
>>> print(eng2sp)
{'one': 'uno'}
```

### 1.23.1 Dictionary as a set of counters

Suppose you are given a string and you want to count how many times each letter appears. There are several ways you could do it:

You could create 26 variables, one for each letter of the alphabet. Then you could traverse the string and, for each character, increment the corresponding counter, probably using a chained conditional.

You could create a list with 26 elements. Then you could convert each character to a number (using the built-in function `ord`), use the number as an index into the list, and increment the appropriate counter.

You could create a dictionary with characters as keys and counters as the corresponding values. The first time you see a character, you would add an item to the dictionary. After that you would increment the value of an existing item.

Each of these options performs the same computation, but each of them implements that computation in a different way.

An implementation is a way of performing a computation; some implementations are better than others. For example, an advantage of the dictionary implementation is that we don't have to know ahead of time which letters appear in the string and we only have to make room for the letters that do appear.

Here is what the code might look like:

```
def histogram(s):
 d = dict()
 for c in s:
 if c not in d:
 d[c] = 1
 else:
 d[c] += 1
 return d
```

The name of the function is histogram, which is a statistical term for a set of counters (or frequencies). The first line of the function creates an empty dictionary. The for loop traverses the string. Each time through the loop, if the character c is not in the dictionary, we create a new item with key c and the initial value 1 (since we have seen this letter once). If c is already in the dictionary we increment d[c].

Here's how it works:

```
>>> h = histogram('brontosaurus')
>>> print(h)
{'a': 1, 'b': 1, 'o': 2, 'n': 1, 's': 2, 'r': 2, 'u': 2, 't': 1}
```

The histogram indicates that the letters 'a' and 'b' appear once; 'o' appears twice, and so on.

## 1.24 6.3

---

### 1.24.1 Dictionaries and lists

Lists can appear as values in a dictionary. For example, if you were given a dictionary that maps from letters to frequencies, you might want to invert it; that is, create a dictionary that maps from frequencies to letters. Since there might be several letters with the same frequency, each value in the inverted dictionary should be a list of letters.

Here is a function that inverts a dictionary:

```
def invert_dict(d):
 inverse = dict()
 for key in d:
 val = d[key]
 if val not in inverse:
 inverse[val] = [key]
 else:
 inverse[val].append(key)
 return inverse
```

Each time through the loop, key gets a key from d and val gets the corresponding value. If val is not in inverse, that means we haven't seen it before, so we create a new item and initialize it with a singleton (a list that contains a single element). Otherwise we have seen this value before, so we append the corresponding key to the list.

Here is an example:

```
>>> hist = histogram('parrot')
>>> print(hist)
{'a': 1, 'p': 1, 'r': 2, 't': 1, 'o': 1}
>>> inverse = invert_dict(hist)
>>> print(inverse)
{1: ['a', 'p', 't'], 2: ['r', 'o']}
```

Lists can be values in a dictionary, as this example shows, but they cannot be keys. Here's what happens if you try:

```
>>> t = [1, 2, 3]
>>> d = dict()
>>> d[t] = 'oops'
Traceback (most recent call last):
```

```
File "<stdin>", line 1, in ?
TypeError: list objects are unhashable
```

A dictionary is implemented using a hash table and that means that the keys have to be hashable.

A hash is a function that takes a value (of any kind) and returns an integer. Dictionaries use these integers, called hash values, to store and look up key-value pairs.

This system works fine if the keys are immutable. But if the keys are mutable, like lists, bad things happen. For example, when you create a key-value pair, Python hashes the key and stores it in the corresponding location. If you modify the key and then hash it again, it would go to a different location. In that case you might have two entries for the same key, or you might not be able to find a key. Either way, the dictionary wouldn't work correctly.

That's why the keys have to be hashable, and why mutable types like lists aren't.

Since lists and dictionaries are mutable, they can't be used as keys, but they can be used as values.

## 1.25 6.4

---

### 1.25.1 Looping and dictionaries

If you use a dictionary in a for statement, it traverses the keys of the dictionary. For example, `print_hist` prints each key and the corresponding value:

```
def print_hist(h):
 for c in h:
 print c, h[c]
```

Here's what the output looks like:

```
>>> h = histogram('parrot')
>>> print_hist(h)
a 1
p 1
r 2
t 1
o 1
```

Again, the keys are in no particular order.

Dictionaries have a method called `keys` that returns the keys of the dictionary, in no particular order, as a list.

## 1.26 Unit 7 - Introduction to Object Oriented Programming

---

## 1.27 7.1

---

### 1.27.1 User-defined types

We have used many of Python's built-in types; now we are going to define a new type. As an example, we will create a type called `Point` that represents a point in two-dimensional space.

In mathematical notation, points are often written in parentheses with a comma separating the coordinates. For example,  $(0,0)$  represents the origin, and  $(x,y)$  represents the point  $x$  units to the right and  $y$  units up from the origin.

There are several ways we might represent points in Python:

- We could store the coordinates separately in two variables, `x` and `y`.
- We could store the coordinates as elements in a list or tuple.
- We could create a new type to represent points as objects.

Creating a new type is (a little) more complicated than the other options, but it has advantages that will be apparent soon.

A user-defined type is also called a class. A class definition looks like this:

```
class Point(object):
 """Represents a point in 2-D space."""
```

This header indicates that the new class is a `Point`, which is a kind of `object`, which is a built-in type.

The body is a docstring that explains what the class is for. You can define variables and functions inside a class definition, but we will get back to that later.

Defining a class named `Point` creates a class object.

```
>>> print(Point)
<class '__main__.Point'=>
```

Because `Point` is defined at the top level, its “full name” is `__main__.Point`.

The class object is like a factory for creating objects. To create a `Point`, you call `Point` as if it were a function.

```
>>> blank = Point()
>>> print(blank)
<__main__.Point instance at 0xb7e9d3ac>
```

The return value is a reference to a `Point` object, which we assign to `blank`. Creating a new object is called instantiation, and the object is an instance of the class.

When you print an instance, Python tells you what class it belongs to and where it is stored in memory (the prefix `0x` means that the following number is in hexadecimal).

## 1.27.2 Attributes

You can assign values to an instance using dot notation:

```
>>> blank.x = 3.0
>>> blank.y = 4.0
```

This syntax is similar to the syntax for selecting a variable from a module, such as `math.pi` or `string.whitespace`. In this case, though, we are assigning values to named elements of an object. These elements are called attributes.

As a noun, AT-trib-ute is pronounced with emphasis on the first syllable, as opposed to a-TRIB-ute, which is a verb.

You can read the value of an attribute using the same syntax:

```
>>> print(blank.y)
4.0
>>> x = blank.x
>>> print(x)
3.0
```

The expression `blank.x` means, “Go to the object `blank` refers to and get the value of `x`.” In this case, we assign that value to a variable named `x`. There is no conflict between the variable `x` and the attribute `x`.

You can use dot notation as part of any expression. For example:

```
>>> print('(%g, %g)' % (blank.x, blank.y)
(3.0, 4.0)
>>> distance = math.sqrt(blank.x**2 + blank.y**2)
>>> print(distance)
5.0
```

You can pass an instance as an argument in the usual way. For example:

```
def print_point(p):
 print('(%g, %g)' % (p.x, p.y))
```

print\_point takes a point as an argument and displays it in mathematical notation. To invoke it, you can pass blank as an argument:

```
>>> print_point(blank)
(3.0, 4.0)
```

Inside the function, p is an alias for blank, so if the function modifies p, blank changes.

### 1.27.2.1 Exercise 6.1

Write a function called distance\_between\_points that takes two Points as arguments and returns the distance between them.

### 1.27.3 Rectangles

Sometimes it is obvious what the attributes of an object should be, but other times you have to make decisions. For example, imagine you are designing a class to represent rectangles. What attributes would you use to specify the location and size of a rectangle? You can ignore angle; to keep things simple, assume that the rectangle is either vertical or horizontal.

There are at least two possibilities:

- You could specify one corner of the rectangle (or the center), the width, and the height.
- You could specify two opposing corners.

At this point it is hard to say whether either is better than the other, so we'll implement the first one, just as an example.

Here is the class definition:

```
class Rectangle(object):
 """Represents a rectangle.

 attributes: width, height, corner.
 """
```

The docstring lists the attributes: width and height are numbers; corner is a Point object that specifies the lower-left corner.

To represent a rectangle, you have to instantiate a Rectangle object and assign values to the attributes:

```
box = Rectangle()
box.width = 100.0
box.height = 200.0
box.corner = Point()
box.corner.x = 0.0
box.corner.y = 0.0
```

The expression box.corner.x means, "Go to the object box refers to and select the attribute named corner; then go to that object and select the attribute named x."

### 1.27.4 Instances as return values

Functions can return instances. For example, find\_center takes a Rectangle as an argument and returns a Point that contains the coordinates of the center of the Rectangle:

```
def find_center(rect):
 p = Point()
```

```
p.x = rect.corner.x + rect.width/2.0
p.y = rect.corner.y + rect.height/2.0
return p
```

Here is an example that passes box as an argument and assigns the resulting Point to center:

```
>>> center = find_center(box)
>>> print_point(center)
(50.0, 100.0)
```

### 1.27.5 Objects are mutable

You can change the state of an object by making an assignment to one of its attributes. For example, to change the size of a rectangle without changing its position, you can modify the values of width and height:

```
box.width = box.width + 50
box.height = box.height + 100
```

You can also write functions that modify objects. For example, `grow_rectangle` takes a Rectangle object and two numbers, `width` and `dheight`, and adds the numbers to the width and height of the rectangle:

```
def grow_rectangle(rect, dwidth, dheight):
 rect.width += dwidth
 rect.height += dheight
```

Here is an example that demonstrates the effect:

```
>>> print(box.width)
100.0
>>> print(box.height)
200.0
>>> grow_rectangle(box, 50, 100)
>>> print(box.width)
150.0
>>> print(box.height)
300.0
```

Inside the function, `rect` is an alias for `box`, so if the function modifies `rect`, `box` changes.

### 1.27.6 Exercise 6.2

Write a function named `move_rectangle` that takes a Rectangle and two numbers named `dx` and `dy`. It should change the location of the rectangle by adding `dx` to the `x` coordinate of `corner` and adding `dy` to the `y` coordinate of `corner`.

## 1.28 7.2

---

### 1.28.1 The `__init__` method

The `__init__` method (short for “initialization”) is a special method that gets invoked when an object is instantiated. Its full name is `__init__` (two underscore characters, followed by `__init__`, and then two more underscores). An `__init__` method for the Time class might look like this:

```
class Time(object):
 def __init__(self, hour=0, minute=0, second=0):
 self.hour = hour
 self.minute = minute
 self.second = second
```

It is common for the parameters of `__init__` to have the same names as the attributes. The statement

```
self.hour = hour
```

stores the value of the parameter hour as an attribute of self.

The parameters are optional, so if you call Time with no arguments, you get the default values.

```
>>> time = Time()
>>> time.print_time()
00:00:00
If you provide one argument, it overrides hour:
>>> time = Time(9)
>>> time.print_time()
09:00:00
If you provide two arguments, they override hour and minute.
>>> time = Time(9, 45)
>>> time.print_time()
09:45:00
```

And if you provide three arguments, they override all three default values.

## 1.28.2 Debugging 6

It is legal to add attributes to objects at any point in the execution of a program, but if you are a stickler for type theory, it is a dubious practice to have objects of the same type with different attribute sets. It is usually a good idea to initialize all of an object's attributes in the `init` method.

If you are not sure whether an object has a particular attribute, you can use the built-in function `hasattr` (see Section 15.7).

Another way to access the attributes of an object is through the special attribute `__dict__`, which is a dictionary that maps attribute names (as strings) and values:

```
>>> p = Point(3, 4)
>>> print(p.__dict__)
{'y': 4, 'x': 3}
```

For purposes of debugging, you might find it useful to keep this function handy:

```
def print_attributes(obj):
 for attr in obj.__dict__:
 print(attr, getattr(obj, attr))
```

`print_attributes` traverses the items in the object's dictionary and prints each attribute name and its corresponding value.

The built-in function `getattr` takes an object and an attribute name (as a string) and returns the attribute's value.

## 1.29 7.3

---

### 1.29.1 Printing objects

We defined a class named `Time` and a function `print_time`:

```
class Time(object):
 """Represents the time of day."""

 def print_time(self):
 print(str(self.hour) + ":" + str(self.minute) + ":" + str(self.second))
```

To call this function, you have to pass a `Time` object as an argument:

```
>>> start = Time()
>>> start.hour = 9
```

```
>>> start.minute = 45
>>> start.second = 00
>>> print_time(start)
09:45:00
```

To make `print_time` a method, all we have to do is move the function definition inside the class definition. Notice the change in indentation.

```
class Time(object):
 def print_time(time):
 print(str(time.hour) + ":" + str(time.minute) + ":" + str(time.second))
```

Now there are two ways to call `print_time`. The first (and less common) way is to use function syntax:

```
>>> Time.print_time(start)
09:45:00
```

In this use of dot notation, `Time` is the name of the class, and `print_time` is the name of the method. `start` is passed as a parameter.

The second (and more concise) way is to use method syntax:

```
>>> start.print_time()
09:45:00
```

In this use of dot notation, `print_time` is the name of the method (again), and `start` is the object the method is invoked on, which is called the subject. Just as the subject of a sentence is what the sentence is about, the subject of a method invocation is what the method is about.

Inside the method, the subject is assigned to the first parameter, so in this case `start` is assigned to `time`.

By convention, the first parameter of a method is called `self`, so it would be more common to write `print_time` like this:

```
class Time(object):
 def print_time(self):
 print(str(self.hour) + ":" + str(self.minute) + ":" + str(self.second))
```

The reason for this convention is an implicit metaphor:

The syntax for a function call, `print_time(start)`, suggests that the function is the active agent. It says something like, "Hey `print_time`! Here's an object for you to print."

In object-oriented programming, the objects are the active agents. A method invocation like `start.print_time()` says "Hey `start`! Please print yourself."

This change in perspective might be more polite, but it is not obvious that it is useful. In the examples we have seen so far, it may not be. But sometimes shifting responsibility from the functions onto the objects makes it possible to write more versatile functions, and makes it easier to maintain and reuse code.

### 1.29.1.1 Exercise 6.3

```
def time_to_int(time):
 minutes = time.hour * 60 + time.minute
 seconds = minutes * 60 + time.second
 return seconds

def int_to_time(seconds):
 time = Time()
 minutes, time.second = divmod(seconds, 60)
 time.hour, time.minute = divmod(minutes, 60)
 return time
```

Write `time_to_int` as a method. It is probably not appropriate to rewrite `int_to_time` as a method; what object you would invoke it on?

## 1.29.2 Another example

Here's a version of increment written as a method:

```
inside class Time:

def increment(self, seconds):
 seconds += self.time_to_int()
 return int_to_time(seconds)
```

This version assumes that `time_to_int` is written as a method, as in Exercise 1. Also, note that it is a pure function, not a modifier.

Here's how you would invoke increment:

```
>>> start.print_time()
09:45:00
>>> end = start.increment(1337)
>>> end.print_time()
10:07:17
```

The subject, `start`, gets assigned to the first parameter, `self`. The argument, `1337`, gets assigned to the second parameter, `seconds`.

This mechanism can be confusing, especially if you make an error. For example, if you invoke `increment` with two arguments, you get:

```
>>> end = start.increment(1337, 460)
TypeError: increment() takes exactly 2 arguments (3 given)
```

The error message is initially confusing, because there are only two arguments in parentheses. But the subject is also considered an argument, so all together that's three.

## 1.29.3 The `__str__` method

`__str__` is a special method, like `__init__`, that is supposed to return a string representation of an object.

For example, here is a `str` method for `Time` objects:

```
inside class Time:

def __str__(self):
 return str(time.hour) + ":" + str(time.minute) + ":" + str(time.second)
```

When you print an object, Python invokes the `str` method:

```
>>> time = Time(9, 45)
>>> print(time)
09:45:00
```

When I write a new class, I almost always start by writing `__init__`, which makes it easier to instantiate objects, and `__str__`, which is useful for debugging.

### 1.29.3.1 Exercise 6.4

Write a `str` method for the `Point` class. Create a `Point` object and print it.

## 1.29.4 Operator overloading

By defining other special methods, you can specify the behavior of operators on user-defined types. For example, if you define a method named `__add__` for the `Time` class, you can use the `+` operator on `Time` objects.

Here is what the definition might look like:

```
inside class Time:

def __add__(self, other):
 seconds = self.time_to_int() + other.time_to_int()
 return int_to_time(seconds)
```

And here is how you could use it:

```
>>> start = Time(9, 45)
>>> duration = Time(1, 35)
>>> print start + duration
11:20:00
```

When you apply the `+` operator to `Time` objects, Python invokes `__add__`. When you print the result, Python invokes `__str__`. So there is quite a lot happening behind the scenes!

Changing the behavior of an operator so that it works with user-defined types is called operator overloading. For every operator in Python there is a corresponding special method, like `__add__`. For more details, see [Python Reference](#).

#### 1.29.4.1 Exercise 6.5

Write an add method for the Point class.

### 1.30 7.4

---

#### 1.30.1 Inheritance

The language feature most often associated with object-oriented programming is inheritance. Inheritance is the ability to define a new class that is a modified version of an existing class.

It is called “inheritance” because the new class inherits the methods of the existing class. Extending this metaphor, the existing class is called the parent and the new class is called the child.

As an example, let’s say we want a class to represent a “hand,” that is, the set of cards held by one player. A hand is similar to a deck: both are made up of a set of cards, and both require operations like adding and removing cards.

A hand is also different from a deck; there are operations we want for hands that don’t make sense for a deck. For example, in poker we might compare two hands to see which one wins. In bridge, we might compute a score for a hand in order to make a bid.

This relationship between classes—similar, but different—lends itself to inheritance.

The definition of a child class is like other class definitions, but the name of the parent class appears in parentheses:

```
class Hand(Deck):
 """Represents a hand of playing cards."""
```

This definition indicates that `Hand` inherits from `Deck`; that means we can use methods like `pop_card` and `add_card` for `Hands` as well as `Decks`.

`Hand` also inherits `__init__` from `Deck`, but it doesn’t really do what we want: instead of populating the hand with 52 new cards, the `init` method for `Hands` should initialize `cards` with an empty list.

If we provide an `__init__` method in the `Hand` class, it overrides the one in the `Deck` class:

```
inside class Hand:
```

```
def __init__(self, label=''):
 self.cards = []
 self.label = label
```

So when you create a Hand, Python invokes this `init` method:

```
>>> hand = Hand('new hand')
>>> print hand.cards
[]
>>> print hand.label
new hand
```

But the other methods are inherited from Deck, so we can use `pop_card` and `add_card` to deal a card:

```
>>> deck = Deck()
>>> card = deck.pop_card()
>>> hand.add_card(card)
>>> print hand
King of Spades
```

A natural next step is to encapsulate this code in a method called `move_cards`:

```
#inside class Deck:

def move_cards(self, hand, num):
 for i in range(num):
 hand.add_card(self.pop_card())
```

`move_cards` takes two arguments, a Hand object and the number of cards to deal. It modifies both `self` and `hand`, and returns None.

In some games, cards are moved from one hand to another, or from a hand back to the deck. You can use `move_cards` for any of these operations: `self` can be either a Deck or a Hand, and `hand`, despite the name, can also be a Deck.

## 1.30.2 Class diagrams

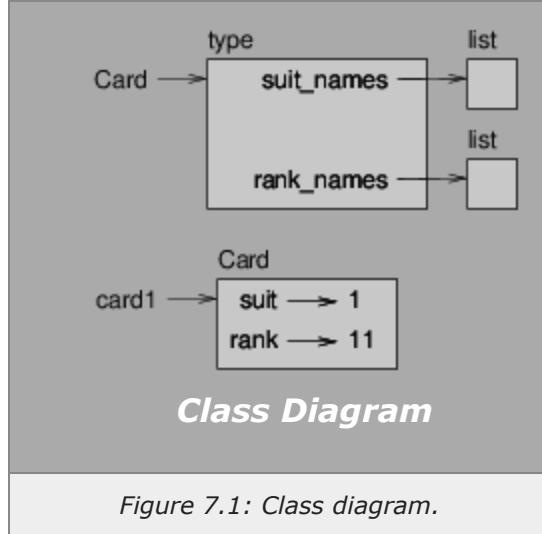
So far we have seen stack diagrams, which show the state of a program, and object diagrams, which show the attributes of an object and their values. These diagrams represent a snapshot in the execution of a program, so they change as the program runs.

They are also highly detailed; for some purposes, too detailed. A class diagram is a more abstract representation of the structure of a program. Instead of showing individual objects, it shows classes and the relationships between them.

There are several kinds of relationship between classes:

- Objects in one class might contain references to objects in another class. For example, each Rectangle contains a reference to a Point, and each Deck contains references to many Cards. This kind of relationship is called HAS-A, as in, “a Rectangle has a Point.”
- One class might inherit from another. This relationship is called IS-A, as in, “a Hand is a kind of a Deck.”
- One class might depend on another in the sense that changes in one class would require changes in the other.

A class diagram is a graphical representation of these relationships. For example, Figure 7.1 shows the relationships between Card, Deck and Hand.



*Figure 7.1: Class diagram.*

The arrow with a hollow triangle head represents an IS-A relationship; in this case it indicates that Hand inherits from Deck.

The standard arrow head represents a HAS-A relationship; in this case a Deck has references to Card objects.

The star (\*) near the arrow head is a multiplicity; it indicates how many Cards a Deck has. A multiplicity can be a simple number, like 52, a range, like 5..7 or a star, which indicates that a Deck can have any number of Cards.

A more detailed diagram might show that a Deck actually contains a list of Cards, but built-in types like `list` and `dict` are usually not included in class diagrams.

### 1.30.2.1 Exercise 7.1

Read `TurtleWorld.py`, `World.py` and `Gui.py` and draw a class diagram that shows the relationships among the classes defined there.

### 1.30.3 Debugging 7

Inheritance can make debugging a challenge because when you invoke a method on an object, you might not know which method will be invoked.

Suppose you are writing a function that works with `Hand` objects. You would like it to work with all kinds of Hands, like `PokerHands`, `BridgeHands`, etc. If you invoke a method like `shuffle`, you might get the one defined in `Deck`, but if any of the subclasses override this method, you'll get that version instead.

Any time you are unsure about the flow of execution through your program, the simplest solution is to add print statements at the beginning of the relevant methods. If `Deck.shuffle` prints a message that says something like Running `Deck.shuffle`, then as the program runs it traces the flow of execution.

As an alternative, you could use this function, which takes an object and a method name (as a string) and returns the class that provides the definition of the method:

```
def find_defining_class(obj, meth_name):
 for ty in type(obj).mro():
 if meth_name in ty.__dict__:
 return ty
```

Here's an example:

```
>>> hand = Hand()
>>> print find_defining_class(hand, 'shuffle')
```

```
<class 'card.Deck'='">>
```

So the shuffle method for this Hand is the one in Deck.

`find_defining_class` uses the `mro()` method to get the list of class objects (types) that will be searched for methods. "MRO" stands for "method resolution order".

Here's a program design suggestion: whenever you override a method, the interface of the new method should be the same as the old. It should take the same parameters, return the same type, and obey the same preconditions and postconditions. If you obey this rule, you will find that any function designed to work with an instance of a superclass, like a `Deck`, will also work with instances of subclasses like a `Hand` or `PokerHand`.

If you violate this rule, your code will collapse like (sorry) a house of cards.

formatted by [Markdeep 1.093](#) 