**Welcome!**

Python is an easy to learn programming language. You can use it to create web apps, games, even a search engine!

**script.py**

print “Welcome to Python!”

**Output:**

Welcome to Python!

None

**Variables**

Creating web apps, games, and search engines all involve storing and working with different types of data. They do so using **variables**. A **variable** stores a piece of data, and gives it a specific name.

For example:

spam = 5

The variable spam now stores the number 5.

1. Set the variable my\_variable equal to the value 10.
2. Click the Save & Submit button to run your code.

**Script.py**

# Write your code below!

my\_variable = 10

**Output:**

None

You will notice that the window says "None" in it when you run the code. This is the "result" of your code, but you can generally ignore it.

**Booleans**

Great! You just stored a number in a variable. Numbers are one data type we use in programming. A second data type is called a **boolean**.

A **boolean** is like a light switch. It can only have two values. Just like a light switch can only be on or off, a boolean can only be True or False.

You can use variables to store booleans like this:

a = True

b = False

Set the following variables to the corresponding values:

1. my\_int to the value 7
2. my\_float to the value 1.23
3. my\_bool to the value True

**Script.py**

# Set the variables to the values listed in the instructions!

my\_int = 7

my\_float = 1.23

my\_bool = True

**Output:**

None

**You've Been Reassigned**

Now you know how to use variables to store values.

Say my\_int = 7. You can change the value of a variable by "reassigning" it, like this:

my\_int = 3

**script.py**

# my\_int is set to 7 below. What do you think

# will happen if we reset it to 3 and print the result?

my\_int = 7

# Change the value of my\_int to 3 on line 8!

my\_int = 3

# Here's some code that will print my\_int to the console:

# The print keyword will be covered in detail soon!

print my\_int

**Output:**

3

None

**Whitespace**

In Python, whitespace is used to structure code. Whitespace is important, so you have to be careful with how you use it.

Instructions

The code on the right is badly formatted. Hit "Save & Submit Code" to see what happens.

You should see an error message. We'll fix it in the next exercise!

**Script.py**

def spam():

eggs = 12

return eggs

print spam()

**Output:**

File "python", line 2  
    eggs = 12  
       ^  
IndentationError: expected an indented block

**Whitespace Means Right Space**

Now let's examine the error from the last lesson:

IndentationError: expected an indented block

You'll get this error whenever your whitespace is off.

Instructions

Properly indent the code with four spaces before eggs on [line 2](javascript:void(0)) and another four before return on [line 3](javascript:void(0)).

You should indent your code with four spaces.

**Script.py**

def spam():

eggs = 12

return eggs

print spam()

**Output:**

12

None

**A Matter of Interpretation**

The window in the top right corner of the page is called the interpreter. The interpreter runs your code line by line, and checks for any errors.

cats = 3

In the above example, we create a variable cats and assign it the value of 3.

Instructions

1. Create a variable called spam and assign it the value of True.
2. Create a variable called eggs and assign it the value of False.

**Script.py**

spam = True

eggs = False

**Output:**

None

**Single Line Comments**

You probably saw us use the # sign a few times in earlier exercises. The # sign is for comments. A comment is a line of text that Python won't try to run as code. It's just for humans to read.

Comments make your program easier to understand. When you look back at your code or others want to collaborate with you, they can read your comments and easily figure out what your code does.

Instructions

Write a comment on [line 1](javascript:void(0)). Make sure it starts with #. It can say anything you like.

**Script.py**

# This is what a comment looks like.

mysterious\_variable = 42

**Output:**

None

**Multi-Line Comments**

The # sign will only comment out a single line. While you could write a multi-line comment, starting each line with #, that can be a pain.

Instead, for multi-line comments, you can include the whole block in a set of triple quotation marks:

"""Sipping from your cup 'til it runneth over,

Holy Grail.

"""

Instructions

Write a multi-line comment in the editor. It can be any text you'd like!

**Math**

Great! Now let's do some math. You can add, subtract, multiply, divide numbers like this

addition = 72 + 23

subtraction = 108 - 204

multiplication = 108 \* 0.5

division = 108 / 9

Instructions

Set the variable count\_to equal to the sum of two big numbers.

**Script.py**

# Set count\_to equal to the sum of two big numbers

count\_to = 25 + 2500

print count\_to

**Output:**

2525

None

**Exponentiation**

All that math can be done on a calculator, so why use Python? Because you can combine math with other data types (e.g. **booleans**) and commands to create useful programs. Calculators just stick to numbers.

Now let's work with exponents.

eight = 2 \*\* 3

In the above example, we create a new variable called eight and set it to 8, or the result of 2 to the power to 3 (2^3).

Notice that we use \*\* instead of \* or the multiplication operator.

Instructions

Create a new variable called eggs and use exponents to set eggs equal 100.

Try raising 10 to the power of 2.

**Script.py**

#Set eggs equal to 100 using exponentiation on line 3!

eggs = 10 \*\* 2

print eggs

**Output:**

100

None

**Modulo**

Our final operator is **modulo**. **Modulo** returns the remainder from a division. So, if you type 3 % 2, it will return 1, because 2 goes into 3 evenly once, with 1 left over.

Instructions

Use modulo to set spam equal to 1. You can use any two numbers that will leave a remainder of 1 to do this.

**Script.py**

#Set spam equal to 1 using modulo on line 3!

spam = 5 % 2

print spam

**Output:**

1

None

**Bringing It All Together**

Nice work! So far, you've learned about:

* **Variables**, which store values for later use
* **Data types**, such as numbers and booleans
* **Whitespace**, which separates statements
* **Comments**, which make your code easier to read
* **Arithmetic operations**, including +, -, \*, /, \*\*, and %

Instructions

Let's put our knowledge to work.

1. Write a single-line comment on [line 1](javascript:void(0)). It can be anything! (Make sure it starts with #)
2. Set the variable monty equal to True.
3. Set another variable python equal to 1.234.
4. Set a third variable monty\_python equal to python squared.

**Script.py**

# Let's start this!

monty = True

python = 1.234

monty\_python = python \*\* 2

**Output:** None

**Tip Calculator**

**The Meal**

Now let's apply the concepts from the previous section to a real world example.

You've finished eating at a restaurant, and received this bill:

* **Cost of meal**: $44.50
* **Restaurant tax**: 6.75%
* **Tip**: 15%

You'll apply the tip to the overall cost of the meal (including tax).

Instructions

First, let's declare the variable meal and assign it the value 44.50.

**Script.py**

# Assign the variable meal the value 44.50 on line 3!

meal = 44.50

**Output:**

None

**The Tax**

Good! Now let's create a variable for the tax percentage.

The tax on your receipt is 6.75%. You'll have to divide 6.75 by 100 in order to get the decimal form of the percentage. (See the Hint if you would like further explanation.)

Instructions

Create the variable tax and set it equal to the decimal value of 6.75%

**Script.py**

meal = 44.50

tax = 6.75/100

**Output:** None

**The Tip**

Nice work! You received good service, so you'd like to leave a 15% tip on top of the cost of the meal, including tax.

Before we compute the tip for your bill, let's set a variable for the tip. Again, we need to get the decimal form of the tip, so we divide 15.0 by 100.

Instructions

Set the variable tip to decimal value of 15% on [line 5](javascript:void(0)).

**Script.py**

# You're almost there! Assign the tip variable on line 5.

meal = 44.50

tax = 0.0675 # 6.75/100

tip = 15.0/100

**Output:**

None

**Reassign in a Single Line**

Okay! We've got the three variables we need to perform our calculation, and we know some arithmetic operators that can help us out.

We saw in Lesson 1 that we can reassign variables. For example, we could say spam = 7, then later change our minds and say spam = 3.

Instructions

On [line 7](javascript:void(0)), reassign meal to the value of itself + itself \* tax. And yes, you're allowed to reassign a variable in terms of itself!

We're only calculating the cost of meal and tax here. We'll get to the tip soon.

**Script.py**

# Reassign meal on line 7!

meal = 44.50

tax = 0.0675

tip = 0.15

meal = meal + meal \* tax

**Output:**

None

**The Total**

Now that meal has the cost of the food plus tax, let's introduce on [line 8](javascript:void(0)) a new variable, total, equal to the new meal + meal \* tip.

The code on [line 10](javascript:void(0)) formats and prints to the console the value of total with exactly two numbers after the decimal. (We'll learn about string formatting, the console, and print in Unit 2!)

Instructions

Assign the variable total to the sum of meal + meal \* tip on [line 8](javascript:void(0)). Now you have the total cost of your meal!

**Script.py**

# Assign the variable total on line 8!

meal = 44.50

tax = 0.0675

tip = 0.15

meal = meal + meal \* tax

total = meal + meal \* tip

print("%.2f" % total)

**Output:**

54.63

None

**Strings & Console Output**

**Strings**

Another useful data type is the **string**. A **string** can contain letters, numbers, and symbols.

name = "Ryan"

age = "19"

food = "cheese"

1. In the above example, we create a variable name and set it to the string value "Ryan".
2. We also set age to "19" and food to "cheese".

Strings need to be within quotes.

Instructions

Create a new variable brian and assign it the string "Hello life!".

**Script.py**

# Set the variable brian on line 3!

brian = "Hello life!"

**Output:**

None

**Practice**

Excellent! Let's get a little practice in with strings.

Instructions

Set the following variables to their respective phrases:

1. Set caesar to "Graham"
2. Set praline to "John"
3. Set viking to "Teresa"

**Script.py**

# Assign your variables below, each on its own line!

caesar = "Graham"

praline = "John"

viking = "Teresa"

# Put your variables above this line

print caesar

print praline

print viking

**Output:**

Graham

John

Teresa

None

**Escaping characters**

There are some characters that cause problems. For example:

'There's a snake in my boot!'

This code breaks because Python thinks the apostrophe in 'There's' ends the string. We can use the backslash to fix the problem, like this:

'There\'s a snake in my boot!'

Instructions

Fix the string in the editor!

'This isn't flying, this is falling with style!'

**Script.py**

# The string below is broken. Fix it using the escape backslash!

'This isn\'t flying, this is falling with style!'

**Output:**

None

**Access by Index**

Great work!

Each character in a string is assigned a number. This number is called the **index**. Check out the diagram in the editor.

c = "cats"[0]

n = "Ryan"[3]

1. In the above example, we create a new variable called c and set it to "c", the character at index zero of the string "cats".
2. Next, we create a new variable called n and set it to "n", the character at index three of the string "Ryan".

In Python, we start counting the index from zero instead of one.

Instructions

On [line 13](javascript:void(0)), assign the variable fifth\_letter equal to the fifth letter of the string "MONTY".

Remember that the fifth letter is not at index 5. Start counting your indices from zero.

**Script.py**

"""

The string "PYTHON" has six characters,

numbered 0 to 5, as shown below:

+---+---+---+---+---+---+

| P | Y | T | H | O | N |

+---+---+---+---+---+---+

0 1 2 3 4 5

So if you wanted "Y", you could just type

"PYTHON"[1] (always start counting from 0!)

"""

fifth\_letter = "MONTY"[4]

print fifth\_letter

**Output:**

Y

None

**String methods**

Great work! Now that we know how to store strings, let's see how we can change them using **string methods**.

**String methods** let you perform specific tasks for strings.

We'll focus on four string methods:

1. len()
2. lower()
3. upper()
4. str()

Let's start with len(), which gets the length (the number of characters) of a string!

Instructions

1. On [line 1](javascript:void(0)), create a variable named parrot and set it to the string "Norwegian Blue". On [line 2](javascript:void(0)), type len(parrot) after the word print, like so: print len(parrot). The output will be the number of letters in "Norwegian Blue"!

**Script.py**

len = "Norwegian Blue"

print len(parrot)

**Output:**

14

None

**lower()**

Well done!

You can use the lower() method to get rid of all the capitalization in your strings. You call lower() like so:

"Ryan".lower()

which will return "ryan".

Instructions

Call lower() on parrot (after print) on [line 3](javascript:void(0)) in the editor.

**Script.py**

parrot = "Norwegian Blue"

print parrot.lower()

**Output:**

norwegian blue

None

**upper()**

Now your string is 100% lower case! A similar method exists to make a string completely upper case.

Instructions

Call upper() on parrot (after print on [line 3](javascript:void(0))) in order to capitalize all the characters in the string!

**Script.py**

parrot = "norwegian blue"

print parrot.upper()

**Output:**

NORWEGIAN BLUE

None

**str()**

Now let's look at str(), which is a little less straightforward. The str() method turns non-strings into strings! For example:

str(2)

would turn 2 into "2".

Instructions

1. Create a variable pi and set it to 3.14 on [line 4](javascript:void(0)).
2. Call str(pi) on [line 5](javascript:void(0)), after print.

**Script.py**

"""Declare and assign your variable on line 4,

then call your method on line 5!"""

pi = 3.14

print str(pi)

**Output:**

3.14

None

**Dot Notation**

Let's take a closer look at why you use len(string) and str(object), but dot notation (such as "String".upper()) for the rest.

lion = "roar"

len(lion)

lion.upper()

Methods that use dot notation only work with strings.

On the other hand, len() and str() can work on other data types.

Instructions

1. On [line 3](javascript:void(0)), call the len() function with the argument ministry.
2. On [line 4](javascript:void(0)), invoke the ministry's .upper() function.

**Script.py**

ministry = "The Ministry of Silly Walks"

print len(ministry)

print ministry.upper()

**Output:**

27

THE MINISTRY OF SILLY WALKS

None

**Printing Strings**

The area where we've been writing our code is called the **editor**.

The **console** (the window in the upper right) is where the results of your code is shown.

print simply displays your code in the console.

Instructions

Print "Monty Python" to the console.

**Script.py**

"""Tell Python to print "Monty Python"

to the console on line 4!"""

print "Monty Python"

**Output:**

Monty Python

None

**Printing Variables**

Great! Now that we've printed strings, let's print variables

Instructions

1. Declare a variable called the\_machine\_goes and assign it the string value "Ping!" on [line 5](javascript:void(0)).
2. Go ahead and print the\_machine\_goes in [line 6](javascript:void(0)).

**Script.py**

"""Assign the string "Ping!" to

the variable the\_machine\_goes on

line 5, then print it out on line 6!"""

the\_machine\_goes = "Ping!"

print the\_machine\_goes

**Output:**

Ping!

None

**String Concatenation**

You know about strings, and you know about arithmetic operators. Now let's combine the two!

print "Life " + "of " + "Brian"

This will print out the phrase Life of Brian.

The + operator between strings will 'add' them together, one after the other. Notice that there are spaces inside the quotation marks after Life and of so that we can make the combined string look like 3 words.

Combining strings together like this is called **concatenation**. Let's try concatenating a few strings together now!

Instructions

Let's give it a try. Print the concatenated strings "Spam ", "and ", "eggs" on [line 3](javascript:void(0)), just like the example above.

Make sure you include the spaces after "Spam " and "and ".

**Script.py**

# Print the concatenation of "Spam and eggs" on line 3!

print "Spam " + "and " + "eggs"

**Output:**

Spam and eggs

None

**Explicit String Conversion**

Sometimes you need to combine a string with something that isn't a string. In order to do that, you have to convert the non-string into a string.

print "I have " + str(2) + " coconuts!"

This will print I have 2 coconuts!.

The str() method converts non-strings into strings. In the above example, you convert the number 2 into a string and then you concatenate the strings together just like in the previous exercise.

Now try it yourself!

Instructions

1. Run the code as-is. You get an error!
2. Use str() to turn 3.14 into a string. Then run the code again.

**Script.py**

# Turn 3.14 into a string on line 3!

print "The value of pi is around " + str(3.14)

**Output:**

The value of pi is around 3.14

None

**String Formatting with %, Part 1**

When you want to print a variable with a string, there is a better method than concatenating strings together.

name = "Mike"

print "Hello %s" % (name)

The % operator after a string is used to combine a string with variables. The % operator will replace a %s in the string with the string variable that comes after it.

Instructions

Take a look at the code in the editor. What do you think it'll do? Click Save & Submit when you think you know.

**Script.py**

string\_1 = "Camelot"

string\_2 = "place"

print "Let's not go to %s. 'Tis a silly %s." % (string\_1, string\_2)

**Output:**

Let's not go to Camelot. 'Tis a silly place.

None

**String Formatting with %, Part 2**

Remember, we used the % operator to replace the %s placeholders with the variables in parentheses.

name = "Mike"

print "Hello %s" % (name)

You need the same number of %s terms in a string as the number of variables in parentheses:

print "The %s who %s %s!" % ("Knights", "say", "Ni")

# This will print "The Knights who say Ni!"

Instructions

Now it's your turn! We have \_\_\_ in the code to show you what you need to change!

1. Inside the string, replace the three \_\_\_ with %s.
2. After the string but before the three variables, replace the final \_\_\_ with a %.
3. Hit *Save & Submit Code*.
4. Answer the questions in the console as they pop up! Type in your answer and hit Enter.

**Script.py**

name = raw\_input("What is your name?")

quest = raw\_input("What is your quest?")

color = raw\_input("What is your favorite color?")

print "Ah, so your name is %s, your quest is %s, " \

"and your favorite color is %s." % (name, quest, color)

**Output:** (asks and answer questions)

What is your name? Matt

What is your quest? Work

What is your favorite color? Black

Ah, so your name is Matt, your quest is Work, and your favorite color is Black.

None

**And Now, For Something Completely Familiar**

Great job! You've learned a lot in this unit, including:

Three ways to create strings

'Alpha'

"Bravo"

str(3)

String methods

len("Charlie")

"Delta".upper()

"Echo".lower()

Printing a string

print "Foxtrot"

Advanced printing techniques

g = "Golf"

h = "Hotel"

print "%s, %s" % (g, h)

Instructions

Let's wrap it all up!

1. On [line 3](javascript:void(0)), create the variable my\_string and set it to any string you'd like.
2. On [line 4](javascript:void(0)), print the length of my\_string.
3. On [line 5](javascript:void(0)), print the .upper() case version of my\_string.

**Script.py**

# Write your code below, starting on line 3!

my\_string = "Hello Everybody"

print len(my\_string)

print my\_string.upper()

**Output:**

15

HELLO EVERYBODY

None

**Date and Time**

**The datetime Library**

A lot of times you want to keep track of when something happened. We can do so in Python using datetime.

Here we'll use datetime to print the date and time in a nice format.

**Script.py**

from datetime import datetime

**Output:**

None

**Getting the Current Date and Time**

We can use a function called datetime.now() to retrieve the current date and time.

from datetime import datetime

print datetime.now()

The first line imports the datetime library so that we can use it.

The second line will print out the current date and time.

Instructions

1. Create a variable called now and store the result of datetime.now() in it.
2. Then, print the value of now.

**Script.py**

from datetime import datetime

now = datetime.now()

print now

**Output:**

2016-03-14 12:00:05.989585

None

**Extracting Information**

Notice how the output looks like 2013-11-25 23:45:14.317454. What if you don't want the entire date and time?

from datetime import datetime

now = datetime.now()

current\_year = now.year

current\_month = now.month

current\_day = now.day

You already have the first two lines.

In the third line, we take the year (and only the year) from the variable now and store it in current\_year.

In the fourth and fifth lines, we store the month and day from now.

Instructions

1. On a new line, print now.year. Make sure you do it after setting the now variable!
2. Then, print out now.month.
3. Finally, print out now.day.

**Script.py**

from datetime import datetime

now = datetime.now()

print now.year

print now.month

print now.day

**Output:**

2016

3

14

None

**Hot Date**

What if we want to print today's date in the following format? mm/dd/yyyy. Let's use string substitution again!

from datetime import datetime

now = datetime.now()

print '%s-%s-%s' % (now.year, now.month, now.day)

# will print: 2014-02-19

Remember that the % operator will fill the %s placeholders in the string on the left with the strings in the parentheses on the right.

In the above example, we print 2014-02-19 (if today is February 19th, 2014), but you are going to print out 02/19/2014.

**Script.py**

from datetime import datetime

now = datetime.now()

print '%s-%s-%s' % (now.year, now.month, now.day)

print '%s/%s/%s' % (now.month, now.day, now.year)

**Output:**

2016-3-14

3/14/2016

None

**Pretty Time**

Nice work! Let's do the same for the hour, minute, and second.

from datetime import datetime

now = datetime.now()

print now.hour

print now.minute

print now.second

In the above example, we just printed the current hour, then the current minute, then the current second.

We can again use the variable now to print the time.

Instructions

Similar to the last exercise, print the current time in the pretty form of hh:mm:ss.

1. Change the string that you are printing so that you have a : character in between the %s placeholders.
2. Change the three things that you are printing from month, day, and year to now.hour, now.minute, and now.second.

**Script.py**

from datetime import datetime

now = datetime.now()

print '%s-%s-%s' % (now.year, now.month, now.day)

print '%s/%s/%s' % (now.month, now.day, now.year)

print '%s:%s:%s' % (now.hour, now.minute, now.second)

**Output:**

2016-3-14

3/14/2016

12:11:31

None

**Grand Finale**

We've managed to print the date and time separately in a very pretty fashion. Let's combine the two!

from datetime import datetime

now = datetime.now()

print '%s/%s/%s' % (now.month, now.day, now.year)

print '%s:%s:%s' % (now.hour, now.minute, now.second)

The example above will print out the date, then on a separate line it will print the time.

Let's print them all on the same line in a single print statement!

Instructions

Print the date and time together in the form: mm/dd/yyyy hh:mm:ss.

To start, change the format string to the left of the % operator.

1. Ensure that it has 6 %s placeholders.
2. Put slashes and colons and a space between the placeholders so that they fit the format above.

Then, change the variables in the parentheses to the right of the % operator.

1. Place the variables so that now.month, now.day, now.year are before now.hour, now.minute, now.second. Make sure that there is a ( before the six and a ) after them.

**Script.py**

from datetime import datetime

now = datetime.now()

print '%s/%s/%s %s:%s:%s' % (now.month, now.day, now.year, now.hour, now.minute, now.second)

**Output:**

3/14/2016 12:15:4

None

**Conditionals & Control Flow**

**Go With the Flow**

Just like in real life, sometimes we'd like our code to be able to make decisions.

The Python programs we've written so far have had one-track minds: they can add two numbers or print something, but they don't have the ability to pick one of these outcomes over the other.

**Control flow** gives us this ability to choose among outcomes based off what else is happening in the program.

Instructions

Check out the code in the editor. You'll see the type of program you'll be able to write once you've mastered control flow. Click Save & Submit to see what happens!

**Script.py**

def clinic():

print "You've just entered the clinic!"

print "Do you take the door on the left or the right?"

answer = raw\_input("Type left or right and hit 'Enter'.").lower()

if answer == "left" or answer == "l":

print "This is the Verbal Abuse Room, you heap of parrot droppings!"

elif answer == "right" or answer == "r":

print "Of course this is the Argument Room, I've told you that already!"

else:

print "You didn't pick left or right! Try again."

clinic()

clinic()

**Output:**

You've just entered the clinic!

Do you take the door on the left or the right?

Type left or right and hit 'Enter'. l

This is the Verbal Abuse Room, you heap of parrot droppings!

None

**Compare Closely!**

Let's start with the simplest aspect of control flow: **comparators**. There are six:

1. Equal to (==)
2. Not equal to (!=)
3. Less than (<)
4. Less than or equal to (<=)
5. Greater than (>)
6. Greater than or equal to (>=)

Comparators check if a value is (or is not) equal to, greater than (or equal to), or less than (or equal to) another value.

Note that == compares whether two things are equal, and = assigns a value to a variable.

Instructions

Set each variable to True or False depending on what you think the result will be. For example, 1 < 2 will be True, because one is less than two.

1. Set bool\_one equal to the result of 17 < 328
2. Set bool\_two equal to the result of 100 == (2 \* 50)
3. Set bool\_three equal to the result of 19 <= 19
   1. Set bool\_four equal to the result of -22 >= -18
   2. Set bool\_five equal to the result of 99 != (98 + 1)

**Script.py**

# Assign True or False as appropriate on the lines below!

# Set this to True if 17 < 328 or to False if it is not.

bool\_one = True # We did this one for you!

# Set this to True if 100 == (2 \* 50) or to False otherwise.

bool\_two = True

# Set this to True if 19 <= 19 or to False if it is not.

bool\_three = True

# Set this to True if -22 >= -18 or to False if it is not.

bool\_four = False

# Set this to True if 99 != (98 + 1) or to False otherwise.

bool\_five = False

**Output:**

None

**Compare... Closelier!**

Excellent! It looks like you're comfortable with basic expressions and comparators.

But what about *extreme* expressions and comparators?

Instructions

Let's run through the comparators again with more complex expressions. Set each variable to True or False depending on what you think the result will be.

1. Set bool\_one to the result of (20 - 10) > 15
2. Set bool\_two to the result of (10 + 17) == 3\*\*16
3. Set bool\_three to the result of 1\*\*2 <= -1
4. Set bool\_four to the result of 40 \* 4 >= -4
5. Set bool\_five to the result of 100 != 10\*\*2

**Script.py**

# Assign True or False as appropriate on the lines below!

# (20 - 10) > 15

bool\_one = False # We did this one for you!

# (10 + 17) == 3\*\*16

# Remember that \*\* can be read as 'to the power of'. 3\*\*16 is about 43 million.

bool\_two = False

# 1\*\*2 <= -1

bool\_three = False

# 40 \* 4 >= -4

bool\_four = True

# 100 != 10\*\*2

bool\_five = False

**Output:**

None

**How the Tables Have Turned**

Comparisons result in either True or False, which are booleans as we learned before in [this exercise](http://www.codecademy.com/courses/introduction-to-python-6WeG3/0/3).

# Make me true!

bool\_one = 3 < 5

Let's switch it up: we'll give the boolean, and you'll write the expression, just like the example above.

Instructions

For each boolean value in the editor, write an expression that evaluates to that value.

Remember, comparators are: ==, !=, >, >=, <, and <=.

Use at least three different ones!

Don't just use True and False! That's cheating!

**Script.py**

# Create comparative statements as appropriate on the lines below!

# Make me true!

bool\_one = 3 < 5 # We already did this one for you!

# Make me false!

bool\_two = 50 == 23\*\*2

# Make me true!

bool\_three = 300 != 3

# Make me false!

bool\_four = 5 > 50

# Make me true!

bool\_five = 22 >= 2

**Output:**

None

**To Be and/or Not to Be**

**Boolean operators** compare statements and result in boolean values. There are three boolean operators:

1. and, which checks if both the statements are True;
2. or, which checks if at least one of the statements is True;
3. not, which gives the opposite of the statement.

We'll go through the operators one by one.

Instructions

Look at the truth table in the editor. Don't worry if you don't completely get it yet—you will by the end of this section!

Click Save & Submit to continue.

"""

Boolean Operators

------------------------ True and True is True

True and False is False

False and True is False

False and False is False

True or True is True

True or False is True

False or True is True

False or False is False

Not True is False

Not False is True

"""

**And**

The boolean operator and returns True when the expressions on both sides of and are true. For instance:

* 1 < 2 and 2 < 3 is True;
* 1 < 2 and 2 > 3 is False.

Instructions

Let's practice with and. Assign each variable to the appropriate boolean value.

1. Set bool\_one equal to the result of False and False
2. Set bool\_two equal to the result of -(-(-(-2))) == -2 and 4 >= 16\*\*0.5
3. Set bool\_three equal to the result of 19 % 4 != 300 / 10 / 10 and False
4. Set bool\_four equal to the result of -(1\*\*2) < 2\*\*0 and 10 % 10 <= 20 - 10 \* 2
5. Set bool\_five equal to the result of True and True

**Script.py**

bool\_one = 3 > 2 and 10 < 5

bool\_two = -(-(-(-2))) == -2 and 4 >= 16\*\*0.5

bool\_three = 19 % 4 != 300 / 10 / 10 and False

bool\_four = -(1\*\*2) < 2\*\*0 and 10 % 10 <= 20 - 10 \* 2

bool\_five = 40 < 50 and 1 != 0

**Output:**

None

**Or**

The boolean operator or returns True when at least one expression on either side of or is true. For example:

* 1 < 2 or 2 > 3 is True;
* 1 > 2 or 2 > 3 is False.

Instructions

Time to practice with or!

1. Set bool\_one equal to the result of 2\*\*3 == 108 % 100 or 'Cleese' == 'King Arthur'
2. Set bool\_two equal to the result of True or False
3. Set bool\_three equal to the result of 100\*\*0.5 >= 50 or False
4. Set bool\_four equal to the result of True or True
5. Set bool\_five equal to the result of 1\*\*100 == 100\*\*1 or 3 \* 2 \* 1 != 3 + 2 + 1

**Script.py**

bool\_one = 2\*\*3 == 108 % 100 or 'Cleese' == 'King Arthur'

bool\_two = len('hello') == 5 or 34 > 60

bool\_three = 100\*\*0.5 >= 50 or False

bool\_four = 2 \* 2 == 4 or 4 < 10

bool\_five = 1\*\*100 == 100\*\*1 or 3 \* 2 \* 1 != 3 + 2 + 1

**Output:**

None

**Not**

The boolean operator not returns True for false statements and False for true statements.

For example:

* not False will evaluate to True, while not 41 > 40 will return False.

Instructions

Let's get some practice with not.

1. Set bool\_one equal to the result of not True
2. Set bool\_two equal to the result of not 3\*\*4 < 4\*\*3
3. Set bool\_three equal to the result of not 10 % 3 <= 10 % 2
4. Set bool\_four equal to the result of not 3\*\*2 + 4\*\*2 != 5\*\*2
5. Set bool\_five equal to the result of not not False

**Script.py**

bool\_one = not True

bool\_two = not 3\*\*4 < 4\*\*3

bool\_three = not 10 % 3 <= 10 % 2

bool\_four = not 3\*\*2 + 4\*\*2 != 5\*\*2

bool\_five = not not False

**Output:**

None

**This and That (or This, But Not That!)**

Boolean operators aren't just evaluated from left to right. Just like with arithmetic operators, there's an order of operations for boolean operators:

1. not is evaluated first;
2. and is evaluated next;
3. or is evaluated last.

For example, True or not False and False returns True. If this isn't clear, look at the Hint.

Parentheses () ensure your expressions are evaluated in the order you want. Anything in parentheses is evaluated as its own unit.

Instructions

Assign True or False as appropriate for bool\_one through bool\_five.

1. Set bool\_one equal to the result of False or not True and True
2. Set bool\_two equal to the result of False and not True or True
3. Set bool\_three equal to the result of True and not (False or False)
4. Set bool\_four equal to the result of not not True or False and not True
5. Set bool\_five equal to the result of False or not (True and True)

**Script.py**

bool\_one = 1 > 100 or not 1 < 100 and 1 == 1

bool\_two = 3\*\*10 < 30 and not 3 < 4 or 20 == 10 \* 2

bool\_three = 1 + 1 < 4 and not (5 < 1 or 6 == 3 - 1 + 2)

bool\_four = not not 10 > 2 or 10 < 5 and not 11 == 1

bool\_five = 4 - 3 > 4 + 3 or not (1 == 2 - 1 and 3 > 2)

**Output:**

None

**Mix 'n' Match**

Great work! We're almost done with boolean operators.

# Make me false

bool\_one = (2 <= 2) and "Alpha" == "Bravo"

Instructions

This time we'll give the expected result, and you'll use some combination of boolean operators to achieve that result.

Remember, the boolean operators are and, or, and not. Use each one at least once!

**Script.py**

# Use boolean expressions as appropriate on the lines below!

# Make me false!

bool\_one = (2 <= 2) and "Alpha" == "Bravo" # We did this one for you!

# Make me true!

bool\_two = 4 \* 4 == 8 + 8 or 3 > 1

# Make me false!

bool\_three = not 5 + 5 >= 2 \* 5 and 666 == "awesome"

# Make me true!

bool\_four = "test" >= "good" and 4 > 1

# Make me true!

bool\_five = 1 + 1 < 3 or "stop" > "go"

**Output:**

None

**Conditional Statement Syntax**

if is a conditional statement that executes some specified code after checking if its expression is True.

Here's an example of if statement syntax:

if 8 < 9:

print "Eight is less than nine!"

In this example, 8 < 9 is the checked expression and print "Eight is less than nine!" is the specified code.

Instructions

If you think the print statement will print to the console, set response equal to 'Y'; otherwise, set response equal to 'N'.

**Script.py**

response = "Y" or "N"

answer = "Left"

if answer == "Left":

print "This is the Verbal Abuse Room, you heap of parrot droppings!"

# Will the above print statement print to the console?

# Set response to 'Y' if you think so, and 'N' if you think not.

**Output:**

This is the Verbal Abuse Room, you heap of parrot droppings!

None

**If You're Having...**

Let's get some practice with if statements. Remember, the syntax looks like this:

if some\_function():

# block line one

# block line two

# et cetera

Looking at the example above, in the event that some\_function() returns True, then the indented block of code after it will be executed. In the event that it returns False, then the indented block will be skipped.

Also, make sure you notice the colons at the end of the if statement. We've added them for you, but they're important.

Instructions

In the editor you'll see two functions. Don't worry about anything unfamiliar. We'll explain soon enough.

1. Replace the underline on [line 2](javascript:void(0)) with an expression that returns True.
2. Replace the underline on [line 6](javascript:void(0)) with an expression that returns True.

If you do it successfully, then both "Success #1" and "Success #2" are printed.

**Script.py**

def using\_control\_once():

if True:

return "Success #1"

def using\_control\_again():

if True:

return "Success #2"

print using\_control\_once()

print using\_control\_again()

**Output:**

Success #1

Success #2

None

**Else Problems, I Feel Bad for You, Son...**

The else statement complements the if statement. An if/else pair says: "If this expression is true, run this indented code block; otherwise, run this code after the else statement."

Unlike if, else doesn't depend on an expression. For example:

if 8 > 9:

print "I don't printed!"

else:

print "I get printed!"

Instructions

Complete the else statements to the right. Note the indentation for each line!

**Script.py**

answer = "'Tis but a scratch!"

def black\_knight():

if answer == "'Tis but a scratch!":

return True

else:

return False # Make sure this returns False

def french\_soldier():

if answer == "Go away, or I shall taunt you a second time!":

return True

else:

return False # Make sure this returns False

**Output:**

None

**I Got 99 Problems, But a Switch Ain't One**

"Elif" is short for "else if." It means exactly what it sounds like: "otherwise, if the following expression is true, do this!"

if 8 > 9:

print "I don't get printed!"

elif 8 < 9:

print "I get printed!"

else:

print "I also don't get printed!"

In the example above, the elif statement is only checked if the original if statement if False.

Instructions

1. On [line 2](javascript:void(0)), fill in the if statement to check *if* answer is greater than 5.
2. On [line 4](javascript:void(0)), fill in the elif so that the function outputs -1 if answer is less than 5.

**Script.py**

def greater\_less\_equal\_5(answer):

if answer > 5:

return 1

elif answer < 5:

return -1

else:

return 0

print greater\_less\_equal\_5(4)

print greater\_less\_equal\_5(5)

print greater\_less\_equal\_5(6)

**Output:**

-1

0

1

None

**The Big If**

Really great work! Here's what you've learned in this unit:

**Comparators**

3 < 4

5 >= 5

10 == 10

12 != 13

**Boolean operators**

True or False

(3 < 4) and (5 >= 5)

this() and not that()

**Conditional statements**

if this\_might\_be\_true():

print "This really is true."

elif that\_might\_be\_true():

print "That is true."

else:

print "None of the above."

Let's get to the grand finale.

Instructions

Write an if statement in the\_flying\_circus(). It must include:

1. if, elif, and else statements;
2. At least one of and, or, or not;
3. A comparator (==, !=, <, <=, >, or >=);
4. Finally, the\_flying\_circus() must return True when evaluated.

Don't forget to include a : after your if statements!

**Script.py**

# Make sure that the\_flying\_circus() returns True

def the\_flying\_circus():

x = 1

if x < 100: # Start coding here!

return True # Don't forget to indent

# the code inside this block!

elif x > 300 \* 1 or x\*\*2 == 25:

return False # Keep going here.

# You'll want to add the else statement, too!

else:

return True

**Output;**

None

**PygLatin**

**Break It Down**

Now let's take what we've learned so far and write a Pig Latin translator.

Pig Latin is a language game, where you move the first letter of the word to the end and add "ay." So "Python" becomes "ythonpay." To write a Pig Latin translator in Python, here are the steps we'll need to take:

1. Ask the user to input a word in English.
2. Make sure the user entered a valid word.
3. Convert the word from English to Pig Latin.
4. Display the translation result.

**Ahoy! (or Should I Say Ahoyay!)**

Let's warm up by printing a welcome message for our translator users.

Instructions

1. Please print the phrase "Pig Latin".

**Script.py**

print "Pig Latin"

**Output:**

Pig Latin

**Input!**

Next, we need to ask the user for input.

name = raw\_input("What's your name?")

print name

In the above example, raw\_input() accepts a string, prints it, and then waits for the user to type something and press Enter (or Return).

In the interpreter, Python will ask:

What's your name? >

Once you type in your name and hit Enter, it will be stored in name.

Instructions

1. On [line 4](javascript:void(0)), use raw\_input("Enter a word:") to ask the user to enter a word. Save the results of raw\_input() in a variable called original.
2. Click Save & Submit Code
3. Type a word in the console window and press Enter (or Return).

**Script.py**

print 'Welcome to the Pig Latin Translator!'

# Start coding here!

original = raw\_input("Enter a word:")

print original

**Output:**

Welcome to the Pig Latin Translator!

Enter a word: Word

Word

None

**Check Yourself!**

Next we need to ensure that the user actually typed something.

empty\_string = ""

if len(empty\_string) > 0:

# Run this block.

# Maybe print something?

else:

# That string must have been empty.

We can check that the user's string actually has characters!

Instructions

Write an if statement that verifies that the string has characters.

1. Add an if statement that checks that len(original) is greater than zero. Don't forget the : at the end of the if statement!
2. If the string actually has some characters in it, print the user's word.
3. Otherwise (i.e. an else: statement), please print "empty".

You'll want to run your code multiple times, testing an empty string and a string with characters. When you're confident your code works, continue to the next exercise.

**Script.py**

print 'Welcome to the Pig Latin Translator!'

# Start coding here!

original = raw\_input("Enter a word:")

if len(original) > 0:

print original

else:

print "empty"

**Output:**

Welcome to the Pig Latin Translator!

Enter a word:

empty

None

**Check Yourself... Some More**

Now we know we have a non-empty string. Let's be even more thorough.

x = "J123"

x.isalpha() # False

In the first line, we create a string with letters and numbers.

The second line then runs the function isalpha() which returns False since the string contains non-letter characters.

Let's make sure the word the user enters contains only alphabetical characters. You can use isalpha() to check this! For example:

Instructions

Use and to add a second condition to your if statement. In addition to your existing check that the string contains characters, you should also use .isalpha() to make sure that it only contains letters.

Don't forget to keep the colon at the end of the if statement!

**Script.py**

print 'Welcome to the Pig Latin Translator!'

# Start coding here!

original = raw\_input("Enter a word:")

if len(original) > 0 and original.isalpha():

print original

else:

print "empty"

**Output:**

Welcome to the Pig Latin Translator!

Enter a word: dvdv1

empty

None

**Ay B C**

Now we can get ready to start translating to Pig Latin! Let's review the rules for translation:

You move the first letter of the word to the end and then append the suffix 'ay'.  
**Example**: python -> ythonpay

Let's create a variable to hold our translation suffix.

Instructions

Create a variable named pyg and set it equal to the suffix 'ay'.

**Script.py**

pyg = 'ay'

**Output:**

None

**Word Up**

Let's simplify things by making the letters in our word lowercase.

the\_string = "Hello"

the\_string = the\_string.lower()

The .lower() function does not modify the string itself, it simply returns a lowercase-version. In the example above, we store the result back into the same variable.

We also need to grab the first letter of the word.

first\_letter = the\_string[0]

second\_letter = the\_string[1]

third\_letter = the\_string[2]

Remember that we start counting from zero, not one, so we access the first letter by asking for [0].

Instructions

Inside your if statement:

1. Create a new variable called word that holds the .lower()-case conversion of original.
2. Create a new variable called first that holds word[0], the first letter of word.

**Script.py**

pyg = 'ay'

original = raw\_input('Enter a word:')

if len(original) > 0 and original.isalpha():

word = original.lower()

first = word[0]

second = word[1]

third = word[2]

print original

else:

print 'empty'

**Output:**

Enter a word: Awesome

awesome

None

**Move it on Back**

Now that we have the first letter stored, we need to add both the letter and the string stored in pyg to the end of the original string.

Remember how to concatenate (i.e. add) strings together?

greeting = "Hello "

name = "D. Y."

welcome = greeting + name

Instructions

On a new line after where you created the first variable:

Create a new variable called new\_word and set it equal to the concatenation of word, first, and pyg.

**Script.py**

pyg = 'ay'

original = raw\_input('Enter a word:')

if len(original) > 0 and original.isalpha():

word = original.lower()

first = word[0]

second = word[1]

third = word[2]

new\_word = word + first + pyg

print original

else:

print 'empty'

**Output:**

Enter a word: GREAT

great

None

**Ending Up**

Well done! However, now we have the first letter showing up both at the beginning and near the end.

s = "Charlie"

print s[0]

# will print "C"

print s[1:4]

# will print "har"

1. First we create a variable s and give it the string "Charlie"
2. Next we access the first letter of "Charlie" using s[0]. Remember letter positions start at 0.
3. Then we access a slice of "Charlie" using s[1:4]. This returns everything from the letter at position 1 up till position 4.

We are going to slice the string just like in the 3rd example above.

Instructions

Set new\_word equal to the slice from the 1st index all the way to the end of new\_word. Use [1:len(new\_word)] to do this.

**Script.py**

pyg = 'ay'

original = raw\_input('Enter a word:')

if len(original) > 0 and original.isalpha():

word = original.lower()

first = word[0]

new\_word = word + first + pyg

new\_word = new\_word[1:len(new\_word)]

print new\_word

else:

print 'empty'

print 'original'

**Output:**

Enter a word: Acceptable

cceptableaay

None

**Testing, Testing, is This Thing On?**

Yay! You should have a fully functioning Pig Latin translator. Test your code thorougly to be sure everything is working smoothly.

You'll also want to take out any print statements you were using to help debug intermediate steps of your code. Now might be a good time to add some comments too! Making sure your code is clean, commented, and fully functional is just as important as writing it in the first place.

Instructions

When you're sure your translator is working just the way you want it, click Save & Submit Code to finish this project.

**Functions**

**What Good are Functions?**

You might have considered the situation where you would like to reuse a piece of code, just with a few different values. Instead of rewriting the whole code, it's much cleaner to define a **function**, which can then be used repeatedly.

Instructions

Check out the code in the editor. If you completed the [Tip Calculator][1] project, you'll remember going through and calculating tax and tip in one chunk of program. Here you can see we've defined two functions: tax to calculate the tax on a bill, and tip to compute the tip.

See how much of the code you understand at first glance (we'll explain it all soon). When you're ready, click Save & Submit to continue.

**Script.py**

def tax(bill):

"""Adds 8% tax to a restaurant bill."""

bill \*= 1.08

print "With tax: %f" % bill

return bill

def tip(bill):

"""Adds 15% tip to a restaurant bill."""

bill \*= 1.15

print "With tip: %f" % bill

return bill

meal\_cost = 100

meal\_with\_tax = tax(meal\_cost)

meal\_with\_tip = tip(meal\_with\_tax)

**Output:**

With tax: 108.000000

With tip: 124.200000

None

**Function Junction**

Functions are defined with three components:

1. The **header**, which includes the def keyword, the name of the function, and any **parameters** the function requires. Here's an example:
2. def hello\_world(): // There are no parameters
3. An optional **comment** that explains what the function does.
4. """Prints 'Hello World!' to the console."""
5. The **body**, which describes the procedures the function carries out. The body is *indented*, just like for conditional statements.
6. print "Hello World!"

Here's the full function pieced together:

def hello\_world():

"""Prints 'Hello World!' to the console."""

print "Hello World!"

Instructions

Go ahead and create a function, spam, that prints the string "Eggs!" to the console. Don't forget to include a comment of your own choosing (enclose it in triple quotes!).

**Script.py**

# Define your spam function starting on line 5. You

# can leave the code on line 11 alone for now--we'll

# explain it soon!

def spam():

""" Will Print out Eggs!"""

print "Eggs!"

# Define the spam function above this line.

spam()

**Output:**

Eggs!

None

**Call and Response**

After defining a function, it must be **called** to be implemented. In the previous exercise, spam() in the last line told the program to look for the function called spam and execute the code inside it.

Instructions

We've set up a function, square. Call it on the number 10 (by putting 10 between the parentheses of square()) on [line 9](javascript:void(0))!

**Script.py**

def square(n):

"""Returns the square of a number."""

squared = n\*\*2

print "%d squared is %d." % (n, squared)

return squared

# Call the square function on line 9! Make sure to

# include the number 10 between the parentheses.

square(10)

**Output:**

10 squared is 100.

None

**Parameters and Arguments**

Let's reexamine the first line that defined square in the previous exercise:

def square(n):

n is a **parameter** of square. A parameter acts as a variable name for a passed in **argument**. With the previous example, we called square with the argument 10. In this instance the function was called, n holds the value 10.

A function can require as many parameters as you'd like, but when you call the function, you should generally pass in a matching number of arguments.

Instructions

Check out the function in the editor, power. It should take two arguments, a base and an exponent, and raise the first to the power of the second. It's currently broken, however, because its parameters are missing.

Replace the \_\_\_s with the parameters base and exponent and call power on a base of 37 and a power of 4.

**Script.py**

def power(base, exponent): # Add your parameters here!

result = base\*\*exponent

print "%d to the power of %d is %d." % (base, exponent, result)

power(37,4) # Add your arguments here!

**Output:**

37 to the power of 4 is 1874161.

None

**Functions Calling Functions**

We've seen functions that can print text or do simple arithmetic, but functions can be much more powerful than that. For example, a function can call another function:

def fun\_one(n):

return n \* 5

def fun\_two(m):

return fun\_one(m) + 7

Instructions

Let's look at the two functions in the editor: one\_good\_turn (which adds 1 to the number it takes in as an argument) and deserves\_another (which adds 2).

Change the body of deserves\_another so that it always adds 2 to the output of one\_good\_turn.

**Script.py**

def one\_good\_turn(n):

return n + 1

def deserves\_another(n):

return one\_good\_turn(n) + 2

**Output:**

None

**Practice Makes Perfect**

Let's create a few more functions just for good measure.

def shout(phrase):

if phrase == phrase.upper():

return "YOU'RE SHOUTING!"

else:

return "Can you speak up?"

shout("I'M INTERESTED IN SHOUTING")

The example above is just there to help you remember how functions are structured.

Don't forget the colon at the end of your function definition!

Instructions

1. First, def a function called cube that takes an argument called number. Don't forget the parentheses and the colon!
2. Make that function return the cube of that number (*i.e.* that number multiplied by itself and multiplied by itself once again).
3. Define a second function called by\_three that takes an argument called number.
4. if that number is divisible by 3, by\_three should call cube(number) and return its result. Otherwise, by\_three should return False.

Don't forget that if and else statements need a : at the end of that line!

**Script.py**

def cube(number):

return number \*\*3

def by\_three(number):

if number % 3 == 0:

return cube(number)

else:

return False

**Output:**

None

**I Know Kung Fu**

Remember import this from the first exercise in this course? That was an example of **import**ing a **module**. A module is a file that contains definitions—including variables and functions—that you can use once it is imported.

Instructions

Before we try any fancy importing, let's see what Python already knows about square roots. On [line 3](javascript:void(0)) in the editor, ask Python to

print sqrt(25)

which we would expect to equal five.

**Script.py**

# Ask Python to print sqrt(25) on line 3.

print sqrt(25)

**Output:**

Traceback (most recent call last):  
  File "python", line 3, in <module>  
NameError: name 'sqrt' is not defined

**Generic Imports**

Did you see that? Python said: "NameError: name 'sqrt' is not defined." Python doesn't know what square roots are—yet.

There is a Python module named math that includes a number of useful variables and functions, and sqrt() is one of those functions. In order to access math, all you need is the import keyword. When you simply import a module this way, it's called a **generic import**.

Instructions

You'll need to do two things here:

1. Type import math on [line 2](javascript:void(0)) in the editor.
2. Insert math. before sqrt() so that it has the form math.sqrt(). This tells Python not only to import math, but to get the sqrt() function from within math.

Then hit Save & Submit to see what Python now knows.

**Script.py**

# Ask Python to print sqrt(25) on line 3.

import math

print math.sqrt(25)

**Output:**

5.0

None

**Function Imports**

Nice work! Now Python knows how to take the square root of a number.

However, we only really needed the sqrt function, and it can be frustrating to have to keep typing math.sqrt().

It's possible to import only certain variables or functions from a given module. Pulling in just a single function from a module is called a **function import**, and it's done with the from keyword:

from module import function

Now you can just type sqrt() to get the square root of a number—no more math.sqrt()!

Instructions

Let's import *only* the sqrt function from math this time. (You don't need the () after sqrt in the from math import sqrt bit.)

**Script.py**

# Import \*just\* the sqrt function from math on line 3!

from math import sqrt

**Output:**

None

**Universal Imports**

Great! We've found a way to handpick the variables and functions we want from modules.

What if we still want all of the variables and functions in a module but don't want to have to constantly type math.?

**Universal import** can handle this for you. The syntax for this is:

from module import \*

Instructions

Use the power of from module import \* to import everything from the math module on [line 3](javascript:void(0)) of the editor.

**Script.py**

# Import \*everything\* from the math module on line 3!

from math import \*

**Output:**

None

**Here Be Dragons**

Universal imports may look great on the surface, but they're not a good idea for one very important reason: they fill your program with a ton of variable and function names without the safety of those names still being associated with the module(s) they came from.

If you have a function of your very own named sqrt and you import math, your function is safe: there is your sqrt and there is math.sqrt. If you do from math import \*, however, you have a problem: namely, two different functions with the exact same name.

Even if your own definitions don't directly conflict with names from imported modules, if you import \* from several modules at once, you won't be able to figure out which variable or function came from where.

For these reasons, it's best to stick with either import module and type module.name or just import specific variables and functions from various modules as needed.

Instructions

The code in the editor will show you everything available in the math module.

Click Save & Submit Code to check it out (you'll see sqrt, along with some other useful things like pi, factorial, and [trigonometric functions](http://en.wikipedia.org/wiki/Trigonometry)).

**Script.py**

import math # Imports the math module

everything = dir(math) # Sets everything to a list of things from math

print everything # Prints 'em all!

**Output:**

['\_\_doc\_\_', '\_\_name\_\_', '\_\_package\_\_', 'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign', 'cos', 'cosh', 'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'hypot', 'isinf', 'isnan', 'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'modf', 'pi', 'pow', 'radians', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'trunc']

None

**On Beyond Strings**

Now that you understand what functions are and how to import modules, let's look at some of the functions that are built in to Python (no modules required!).

You already know about some of the built-in functions we've used with strings, such as .upper(), .lower(), str(), and len(). These are great for doing work with strings, but what about something a little more analytic?

Instructions

What do you think the code in the editor will do? Click Save & Submit Code when you think you have an idea.

**Script.py**

def biggest\_number(\*args):

print max(args)

return max(args)

def smallest\_number(\*args):

print min(args)

return min(args)

def distance\_from\_zero(arg):

print abs(arg)

return abs(arg)

biggest\_number(-10, -5, 5, 10)

smallest\_number(-10, -5, 5, 10)

distance\_from\_zero(-10)

**Output:**

10

-10

10

None

**max()**

The max() function takes any number of arguments and returns the largest one. ("Largest" can have odd definitions here, so it's best to use max() on integers and floats, where the results are straightforward, and not on other objects, like strings.)

For example, max(1,2,3) will return 3 (the largest number in the set of arguments).

Instructions

Try out the max() function on [line 3](javascript:void(0)) of the editor. You can provide any number of integer or float arguments to max().

**Script.py**

# Set maximum to the max value of any set of numbers on line 3!

maximum = max(2, .33, 25, -3)

print maximum

**Output:**

25

None

**min()**

min() then returns the smallest of a given series of arguments.

Instructions

Go ahead and set minimum equal to the min() of any set of integers or floats you'd like.

**Script.py**

# Set minimum to the min value of any set of numbers on line 3!

minimum = min(2, .35, 25, -3)

print minimum

**Output:**

-3

None

**abs()**

The abs() function returns the **absolute value** of the number it takes as an argument—that is, that number's distance from 0 on an imagined number line. For instance, 3 and -3 both have the same absolute value: 3. The abs() function always returns a positive value, and unlike max() and min(), it only takes a single number.

Instructions

Set absolute equal to the absolute value of -42 on [line 2](javascript:void(0)).

**Script.py**

absolute = abs(-42)

print absolute

**Output:**

42

None

**type()**

Finally, the type() function returns the **type** of the data it receives as an argument. If you ask Python to do the following:

print type(42)

print type(4.2)

print type('spam')

Python will output:

<type 'int'>

<type 'float'>

<type 'str'>

Instructions

Have Python print out the type of an int, a float, and a str string in the editor. You can pick any values on which to call type(), so long as they produce one of each.

**Script.py**

# Print out the types of an integer, a float,

# and a string on separate lines below.

print type(45)

print type(6.6)

print type('Hello')

**Output:**

<type 'int'>

<type 'float'>

<type 'str'>

None

**Review: Functions**

Okay! Let's review functions.

def speak(message):

return message

if happy():

speak("I'm happy!")

elif sad():

speak("I'm sad.")

else:

speak("I don't know what I'm feeling.")

Again, the example code above is just there for your reference!

Instructions

1. First, def a function, shut\_down, that takes one argument s. Don't forget the parentheses or the colon!
2. Then, if the shut\_down function receives an s equal to "yes", it should return "Shutting down"
3. Alternatively, elif s is equal to "no", then the function should return "Shutdown aborted".
4. Finally, if shut\_down gets anything other than those inputs, the function should return "Sorry"

**Script.py**

def shut\_down(s):

if s == "yes":

return "Shutting down"

elif s == "no":

return "Shutdown aborted"

else:

return "Sorry"

print shut\_down("yes")

**Output:**

Shutting down

None

**Review: Modules**

Good work! Now let's see what you remember about importing modules (and, specifically, what's available in the math module).

Instructions

Import the math module in whatever way you prefer. Call its sqrt function on the number 13689 and print that value to the console.

**Script.py**

from math import sqrt

print sqrt(13689)

**Output:**

117.0

None

**Review: Built-In Functions**

Perfect! Last but not least, let's review the built-in functions you've learned about in this lesson.

def is\_numeric(num):

return type(num) == int or type(num) == float:

max(2, 3, 4) # 4

min(2, 3, 4) # 2

abs(2) # 2

abs(-2) # 2

Instructions

1. First, def a function called distance\_from\_zero, with one argument (choose any argument name you like).
2. If the type of the argument is either int or float, the function should return the absolute value of the function input.
3. Otherwise, the function should return "Nope"

**Script.py**

def distance\_from\_zero(x):

if type(x) == int or type(x) == float:

return abs(x)

else:

return "Nope"

print distance\_from\_zero(20)

**Output:**

20

None

**Python Lists and Dictionaries**

**Introduction to Lists**

Lists are a **datatype** you can use to store a collection of different pieces of information as a sequence under a single variable name. (Datatypes you've already learned about include strings, numbers, and booleans.)

You can assign items to a list with an expression of the form

list\_name = [item\_1, item\_2]

with the items in between brackets. A list can also be empty: empty\_list = [].

Lists are very similar to strings, but there are a few key differences.

Instructions

The list zoo\_animals has three items (check them out on [line 1](javascript:void(0))). Go ahead and add a fourth! Just enter the name of your favorite animal (as a "string") on [line 1](javascript:void(0)), after the final comma but before the closing ].

**Script.py**

zoo\_animals = ["pangolin", "cassowary", "sloth", "bear"];

# One animal is missing!

if len(zoo\_animals) > 3:

print "The first animal at the zoo is the " + zoo\_animals[0]

print "The second animal at the zoo is the " + zoo\_animals[1]

print "The third animal at the zoo is the " + zoo\_animals[2]

print "The fourth animal at the zoo is the " + zoo\_animals[3]

**Output:**

The first animal at the zoo is the pangolin

The second animal at the zoo is the cassowary

The third animal at the zoo is the sloth

The fourth animal at the zoo is the bear

None

**Access by Index**

You can access an individual item on the list by its **index**. An index is like an address that identifies the item's place in the list. The index appears directly after the list name, in between brackets, like this: list\_name[index].

**List indices begin with 0, not 1!** You access the first item in a list like this: list\_name[0]. The second item in a list is at index 1: list\_name[1]. Computer scientists love to start counting from zero.

Instructions

Write a statement that prints the result of adding the second and fourth items of the list. Make sure to access the list by index!

**Script.py**

numbers = [5, 6, 7, 8]

print "Adding the numbers at indices 0 and 2..."

print numbers[0] + numbers[2]

print "Adding the numbers at indices 1 and 3..."

# Your code here!

print numbers[1] + numbers[3]

**Output:**

Adding the numbers at indices 0 and 2...

12

Adding the numbers at indices 1 and 3...

14

None

**New Neighbors**

A list index behaves like any other variable name! It can be used to access as well as assign values.

You saw how to access a list index like this:

zoo\_animals[0]

# Gets the value "pangolin"

You can see how assignment works on [line 5](javascript:void(0)):

zoo\_animals[2] = "hyena"

# Changes "sloth" to "hyena"

Instructions

Write an assignment statement that will replace the item that currently holds the value "tiger" with another animal (as a string). It can be any animal you like.

**Script.py**

zoo\_animals = ["pangolin", "cassowary", "sloth", "tiger"]

# Last night our zoo's sloth brutally attacked

#the poor tiger and ate it whole.

# The ferocious sloth has been replaced by a friendly hyena.

zoo\_animals[2] = "hyena"

# What shall fill the void left by our dear departed tiger?

# Your code here!

zoo\_animals[3] = "otter"

**Output:**

None

**Late Arrivals & List Length**

A list doesn't have to have a fixed length. You can add items to the end of a list any time you like!

letters = ['a', 'b', 'c']

letters.append('d')

print len(letters)

print letters

1. In the above example, we first create a list called letters.
2. Then, we add the string 'd' to the end of the letters list.
3. Next, we print out 4, the length of the letters list.
4. Finally, we print out ['a', 'b', 'c', 'd'].

Instructions

1. On [lines 5](javascript:void(0)), 6, and 7, append three more items to the suitcase list, just like the second line of the example above. (Maybe bring a bathing suit?)
2. Then, set list\_length equal to the length of the suitcase list.

**Script.py**

suitcase = []

suitcase.append("sunglasses")

# Your code here!

suitcase.append("underwear")

suitcase.append("shirt")

suitcase.append("pants")

list\_length = len(suitcase) # Set this to the length of suitcase

print "There are %d items in the suitcase." % (list\_length)

print suitcase

**Output:**

There are 4 items in the suitcase.

['sunglasses', 'underwear', 'shirt', 'pants']

None

**List Slicing**

Sometimes, you only want to access a portion of a list.

letters = ['a', 'b', 'c', 'd', 'e']

slice = letters[1:3]

print slice

print letters

1. In the above example, we first create a list called letters.
2. Then, we take a subsection and store it in the slice list. We start at the index before the colon and continue up to but not including the index after the colon.
3. Next, we print out ['b', 'c']. Remember that we start counting indices from 0 and that we stopped before index 3.
4. Finally, we print out ['a', 'b', 'c', 'd', 'e'], just to show that we did not modify the original letters list.

Instructions

1. On [line 4](javascript:void(0)), create a list called middle containing only the two middle items from suitcase.
2. On [line 5](javascript:void(0)), create a list called last made up only of the last two items from suitcase.

**Script.py**

suitcase = ["sunglasses", "hat", "passport", "laptop", "suit", "shoes"]

first = suitcase[0:2] # The first and second items (index zero and one)

middle = suitcase[2:4] # Third and fourth items (index two and three)

last = suitcase[4:] # The last two items (index four and five)

**Output:**

None

**Slicing Lists and Strings**

You can slice a string exactly like a list! In fact, you can think of strings as lists of characters: each character is a sequential item in the list, starting from index 0.

my\_list[:2]

# Grabs the first two items

my\_list[3:]

# Grabs the fourth through last items

If your list slice includes the very first or last item in a list (or a string), the index for that item doesn't have to be included.

Instructions

1. Assign to dog a slice of animals from index 3 up until *but not including* index 6.
2. Assign to frog a slice of animals from index 6 until the end of the string.

**Script.py**

animals = "catdogfrog"

cat = animals[:3] # The first three characters of animals

dog = animals[3:6] # The fourth through sixth characters

frog = animals[6:] # From the seventh character to the end

**Output:**

None

**Maintaining Order**

Sometimes you need to search for an item in a list.

animals = ["ant", "bat", "cat"]

print animals.index("bat")

1. First, we create a list called animals with three strings.
2. Then, we print the first index that contains the string "bat", which will print 1.

We can also insert items into a list.

animals.insert(1, "dog")

print animals

1. We insert "dog" at index 1, which moves everything down by 1.
2. We print out ["ant", "dog", "bat", "cat"]

Instructions

1. Use the .index(item) function to find the index of "duck". Assign that result to a variable called duck\_index.
2. Then .insert(index, item) the string "cobra" at that index.

**Script.py**

animals = ["aardvark", "badger", "duck", "emu", "fennec fox"]

duck\_index = animals.index("duck") # Use index() to find "duck"

# Your code here!

animals.insert(duck\_index, "cobra")

print animals # Observe what prints after the insert operation

**Output:**

['aardvark', 'badger', 'cobra', 'duck', 'emu', 'fennec fox']

None

**For One and All**

If you want to do something with every item in the list, you can use a for loop. If you've learned about for loops in JavaScript, pay close attention! They're different in Python.

for variable in list\_name:

# Do stuff!

A variable name follows the for keyword; it will be assigned the value of each list item in turn.

Then in list\_name designates list\_name as the list the loop will work on. The line ends with a colon (:) and the indented code that follows it will be executed once per item in the list.

Instructions

Write a statement in the indented part of the for-loop that prints a number equal to 2 \* number for every list item.

**Script.py**

my\_list = [1,9,3,8,5,7]

for number in my\_list:

# Your code here

print 2 \* number

**Output:**

2

18

6

16

10

14

None

**More with 'for'**

If your list is a jumbled mess, you may need to sort() it.

animals = ["cat", "ant", "bat"]

animals.sort()

for animal in animals:

print animal

1. First, we create a list called animals with three strings. The strings are not in alphabetical order.
2. Then, we sort animals into alphabetical order. Note that .sort() modifies the list rather than returning a new list.
3. Then, for each item in animals, we print that item out as "ant", "bat", "cat" on their own line each.

Instructions

1. Write a for-loop that iterates over start\_list and .append()s each number squared (x \*\* 2) to square\_list.
2. Then sort square\_list!

**Script.py**

start\_list = [5, 3, 1, 2, 4]

square\_list = []

# Your code here!

for x in start\_list:

square\_list.append(x \*\* 2)

square\_list.sort()

print square\_list

**Output:**

[1, 4, 9, 16, 25]

None

**This Next Part is Key**

A dictionary is similar to a list, but you access values by looking up a **key** instead of an index. A key can be any string or number. Dictionaries are enclosed in curly braces, like so:

d = {'key1' : 1, 'key2' : 2, 'key3' : 3}

This is a dictionary called d with three **key-value pairs**. The key 'key1' points to the value 1, 'key2' to 2, and so on.

Dictionaries are great for things like phone books (pairing a name with a phone number), login pages (pairing an e-mail address with a username), and more!

Instructions

Print the values stored under the 'Sloth' and 'Burmese Python' keys. Accessing dictionary values by key is just like accessing list values by index:

residents['Puffin']

# Gets the value 104

**Script.py**

# Assigning a dictionary with three key-value pairs to residents:

residents = {'Puffin' : 104, 'Sloth' : 105, 'Burmese Python' : 106}

print residents['Puffin'] # Prints Puffin's room number

# Your code here!

print residents['Sloth']

print residents['Burmese Python']

**Output:**

104

105

106

None

**New Entries**

Like Lists, Dictionaries are "mutable". This means they can be changed after they are created. One advantage of this is that we can add new key/value pairs to the dictionary after it is created like so:

dict\_name[new\_key] = new\_value

An empty pair of curly braces {} is an empty dictionary, just like an empty pair of [] is an empty list.

The length len() of a dictionary is the number of key-value pairs it has. Each pair counts only once, even if the value is a list. (That's right: you can put lists *inside* dictionaries!)

Instructions

Add at least three more key-value pairs to the menu variable, with the dish name (as a "string") for the key and the price (a float or integer) as the value. Here's an example:

menu['Spam'] = 2.50

**Script.py**

menu = {} # Empty dictionary

menu['Chicken Alfredo'] = 14.50 # Adding new key-value pair

print menu['Chicken Alfredo']

# Your code here: Add some dish-price pairs to menu!

menu['Spaghetti'] = 12.50

menu['Pizza'] = 14.50

menu['Lasagna'] = 11.25

print "There are " + str(len(menu)) + " items on the menu."

print menu

**Output:**

14.5

There are 4 items on the menu.

{'Lasagna': 11.25, 'Chicken Alfredo': 14.5, 'Spaghetti': 12.5, 'Pizza': 14.5}

None

**Changing Your Mind**

Because dictionaries are mutable, they can be changed in many ways. Items can be removed from a dictionary with the del command:

del dict\_name[key\_name]

will remove the key key\_name and its associated value from the dictionary.

A new value can be associated with a key by assigning a value to the key, like so:

dict\_name[key] = new\_value

Instructions

Delete the 'Sloth' and 'Bengal Tiger' items from zoo\_animals using del.

Set the value associated with 'Rockhopper Penguin' to anything other than 'Arctic Exhibit'.

**Script.py**

# key - animal\_name : value - location

zoo\_animals = { 'Unicorn' : 'Cotton Candy House',

'Sloth' : 'Rainforest Exhibit',

'Bengal Tiger' : 'Jungle House',

'Atlantic Puffin' : 'Arctic Exhibit',

'Rockhopper Penguin' : 'Arctic Exhibit'}

# A dictionary (or list) declaration may break across multiple lines

# Removing the 'Unicorn' entry. (Unicorns are incredibly expensive.)

del zoo\_animals['Unicorn']

# Your code here!

del zoo\_animals['Sloth']

del zoo\_animals['Bengal Tiger']

zoo\_animals['Rockhopper Penguin'] = 'North Pole Exhibit'

print zoo\_animals

**Output:**

{'Atlantic Puffin': 'Arctic Exhibit', 'Rockhopper Penguin': 'North Pole Exhibit'}

None

**Remove a Few Things**

Sometimes you need to remove something from a list.

beatles = ["john","paul","george","ringo","stuart"]

beatles.remove("stuart")

print beatles

>> ["john","paul","george","ringo"]

1. We create a list called beatles with 5 strings.
2. Then, we remove the first item from beatles that matches the string "stuart". Note that .remove(item) does not return anything.
3. Finally, we print out that list just to see that "stuart" was actually removed.

Instructions

Remove 'dagger' from the list of items stored in the backpack variable.

**Script.py**

backpack = ['xylophone', 'dagger', 'tent', 'bread loaf']

backpack.remove('dagger')

**Output:**

None

**It's Dangerous to Go Alone! Take This**

Let's go over a few last notes about **dictionaries**

my\_dict = {

"fish": ["c", "a", "r", "p"],

"cash": -4483,

"luck": "good"

}

print my\_dict["fish"][0]

1. In the example above, we created a dictionary that holds many types of values.
2. The key "fish" has a list, the key "cash" has an int, and the key "luck" has a string.
3. Finally, we print the letter 'c'. When we access a value in the dictionary like my\_dict["fish"], we have direct access to that value. So we can access the item at index '0' in the list stored by the key "fish"

Instructions

1. Add a key to inventory called 'pocket'
2. Set the value of 'pocket' to be a list consisting of the strings 'seashell', 'strange berry', and 'lint'
3. .sort() the items in the list stored under the 'backpack' key
4. Then .remove('dagger') from the list of items stored under the 'backpack' key
5. Add 50 to the number stored under the 'gold' key

**Script.py**

inventory = {

'gold' : 500,

'pouch' : ['flint', 'twine', 'gemstone'], # Assigned a new list to 'pouch' key

'backpack' : ['xylophone','dagger', 'bedroll','bread loaf']

}

# Adding a key 'burlap bag' and assigning a list to it

inventory['burlap bag'] = ['apple', 'small ruby', 'three-toed sloth']

# Sorting the list found under the key 'pouch'

inventory['pouch'].sort()

# Your code here

inventory['pocket'] = ['seashell', 'strange berry', 'lint']

inventory['backpack'].sort()

inventory['backpack'].remove('dagger')

inventory['gold'] = 500 + 50

**Output:**

None

**Student Becomes the Teacher**

**Lesson Number One**

Welcome to this "Challenge Course". Until now we've been leading you by the hand and working on some short and relatively easy projects. This is a **challenge** so be ready. We have faith in you!

We’re going to switch it up a bit and allow you to be the teacher of your own class. Make a gradebook for all of your students.

animal\_sounds = {

"cat": ["meow", "purr"],

"dog": ["woof", "bark"],

"fox": [],

}

print animal\_sounds["cat"]

The example above is just to remind you how to create a dictionary and then to access the item stored by the "cat" key.

Instructions

1. Create three dictionaries: lloyd, alice, and tyler.
2. Give each dictionary the keys "name", "homework", "quizzes", and "tests".
3. Have the "name" key be the name of the student (that is, lloyd's name should be "Lloyd") and the other keys should be an empty list. (We'll fill in these lists soon!)

Script.py

lloyd = {

"name": "Lloyd",

"homework":[],

"quizzes":[],

"tests":[]

}

alice = {

"name": "Alice",

"homework":[],

"quizzes":[],

"tests":[]

}

tyler = {

"name": "Tyler",

"homework":[],

"quizzes":[],

"tests":[]

}

**Output:**

None

**What's the Score?**

Great work!

Instructions

Now fill out your lloyd dictionary with the appropriate scores. To save you some time, we've filled out the rest for you.

Homework: 90.0, 97.0, 75.0, 92.0  
Quizzes: 88.0, 40.0, 94.0  
Test Scores: 75.0, 90.0

**Make sure to include the decimal points so your grades are stored as floats!** This will be important later.

**Script.py**

lloyd = {

"name": "Lloyd",

"homework": [],

"quizzes": [],

"tests": []

}

alice = {

"name": "Alice",

"homework": [100.0, 92.0, 98.0, 100.0],

"quizzes": [82.0, 83.0, 91.0],

"tests": [89.0, 97.0]

}

tyler = {

"name": "Tyler",

"homework": [0.0, 87.0, 75.0, 22.0],

"quizzes": [0.0, 75.0, 78.0],

"tests": [100.0, 100.0]

}

lloyd['homework'] = [90.0, 97.0, 75.0, 92.0]

lloyd['quizzes'] = [88.0, 40.0, 94.0]

lloyd['tests'] = [75.0, 90.0]

**Output:**

None

**Put It Together**

Now lets put the three dictionaries in a list together.

my\_list = [1, 2, 3]

The above example is just a reminder on how to create a list. Afterwards, my\_list contains 1, 2, and 3.

Instructions

Below your code, create a list called students that contains lloyd, alice, and tyler.

**Script.py**

lloyd = {

"name": "Lloyd",

"homework": [],

"quizzes": [],

"tests": []

}

alice = {

"name": "Alice",

"homework": [100.0, 92.0, 98.0, 100.0],

"quizzes": [82.0, 83.0, 91.0],

"tests": [89.0, 97.0]

}

tyler = {

"name": "Tyler",

"homework": [0.0, 87.0, 75.0, 22.0],

"quizzes": [0.0, 75.0, 78.0],

"tests": [100.0, 100.0]

}

lloyd['homework'] = [90.0, 97.0, 75.0, 92.0]

lloyd['quizzes'] = [88.0, 40.0, 94.0]

lloyd['tests'] = [75.0, 90.0]

students = [lloyd,alice,tyler]

**Output:**

None

**For the Record**

Excellent. Now you need a hard copy document with all of your students' grades.

animal\_sounds = {

"cat": ["meow", "purr"],

"dog": ["woof", "bark"],

"fox": [],

}

print animal\_sounds["cat"]

The example above is just to remind you how to create a dictionary and then to access the item stored by the "cat" key.

Instructions

for each student in your students list, print out that student's data, as follows:

1. print the student's name
2. print the student's homework
3. print the student's quizzes
4. print the student's tests

**Script.py**

lloyd = {

"name": "Lloyd",

"homework": [],

"quizzes": [],

"tests": []

}

alice = {

"name": "Alice",

"homework": [100.0, 92.0, 98.0, 100.0],

"quizzes": [82.0, 83.0, 91.0],

"tests": [89.0, 97.0]

}

tyler = {

"name": "Tyler",

"homework": [0.0, 87.0, 75.0, 22.0],

"quizzes": [0.0, 75.0, 78.0],

"tests": [100.0, 100.0]

}

lloyd['homework'] = [90.0, 97.0, 75.0, 92.0]

lloyd['quizzes'] = [88.0, 40.0, 94.0]

lloyd['tests'] = [75.0, 90.0]

students = [lloyd,alice,tyler]

for student in students:

print student["name"]

print student["homework"]

print student["quizzes"]

print student["tests"]

**Output:**

Lloyd

[90.0, 97.0, 75.0, 92.0]

[88.0, 40.0, 94.0]

[75.0, 90.0]

Alice

[100.0, 92.0, 98.0, 100.0]

[82.0, 83.0, 91.0]

[89.0, 97.0]

Tyler

[0.0, 87.0, 75.0, 22.0]

[0.0, 75.0, 78.0]

[100.0, 100.0]

None

**It's Okay to be Average**

When teaching a class, it's important to take the students' averages in order to assign grades.

5 / 2

# 2

5.0 / 2

# 2.5

float(5) / 2

# 2.5

The above example is a reminder of how division works in Python.

1. When you divide an integer by another integer, the result is always an integer (rounded down, if needed).
2. When you divide a float by an integer, the result is always a float.
3. To divide two integers and end up with a float, you must first use float() to convert one of the integers to a float.

Instructions

Write a function average that takes a list of numbers and returns the average.

1. Define a function called average that has one argument, numbers.
2. Inside that function, call the built-in sum() function with the numbers list as a parameter. Store the result in a variable called total.
3. Like the example above, use float() to convert total and store the result in total.
4. Divide total by the length of the numbers list. Use the built-in len() function to calculate that.
5. Return that result.

**Script.py**

lloyd = {

"name": "Lloyd",

"homework": [90.0, 97.0, 75.0, 92.0],

"quizzes": [88.0, 40.0, 94.0],

"tests": [75.0, 90.0]

}

alice = {

"name": "Alice",

"homework": [100.0, 92.0, 98.0, 100.0],

"quizzes": [82.0, 83.0, 91.0],

"tests": [89.0, 97.0]

}

tyler = {

"name": "Tyler",

"homework": [0.0, 87.0, 75.0, 22.0],

"quizzes": [0.0, 75.0, 78.0],

"tests": [100.0, 100.0]

}

# Add your function below!

def average(numbers):

total = sum(numbers)

total = float(total)

total = total / len(numbers)

return total

**Output:**

None

**Just Weight and See**

Great! Now we need to compute a student’s average using weighted averages.

cost = {

"apples": [3.5, 2.4, 2.3],

"bananas": [1.2, 1.8],

}

return 0.9 \* average(cost["apples"]) + \

0.1 \* average(cost["bananas"])

1. In the above example, we create a dictionary called cost that contains the costs of some fruit.
2. Then, we calculate the average cost of apples and the average cost of bananas. Since we like apples much more than we like bananas, we weight the average cost of apples by 90% and the average cost of bananas by 10%.

The \ character is a *continuation character*. The following line is considered a *continuation* of the current line.

Instructions

Write a function called get\_average that takes a student dictionary (like lloyd, alice, or tyler) as input and returns his/her weighted average.

1. Define a function called get\_average that takes one argument called student.
2. Make a variable homework that stores the average() of student["homework"].
3. Repeat step **2** for "quizzes" and "tests".
4. Multiply the 3 averages by their weights and return the sum of those three. Homework is 10%, quizzes are 30% and tests are 60%.

**Script.py**

lloyd = {

"name": "Lloyd",

"homework": [90.0, 97.0, 75.0, 92.0],

"quizzes": [88.0, 40.0, 94.0],

"tests": [75.0, 90.0]

}

alice = {

"name": "Alice",

"homework": [100.0, 92.0, 98.0, 100.0],

"quizzes": [82.0, 83.0, 91.0],

"tests": [89.0, 97.0]

}

tyler = {

"name": "Tyler",

"homework": [0.0, 87.0, 75.0, 22.0],

"quizzes": [0.0, 75.0, 78.0],

"tests": [100.0, 100.0]

}

# Add your function below!

def average(numbers):

total = sum(numbers)

total = float(total)

total = total / len(numbers)

return total

def get\_average(student):

homework = average(student["homework"])

quizzes = average(student["quizzes"])

tests = average(student["tests"])

return 0.1 \* homework + 0.3 \* quizzes + 0.6 \* tests

**Output**:

None

**Sending a Letter**

Great work!

Now let's write a get\_letter\_grade function that takes a number score as input and returns a string with the letter grade that that student should receive.

Instructions

1. Define a new function called get\_letter\_grade that has one argument called score. Expect score to be a number.
2. Inside your function, test score using a chain of if: / elif: / else: statements, like so:

If score is 90 or above: return "A"  
Else if score is 80 or above: return "B"  
Else if score is 70 or above: return "C"  
Else if score is 60 or above: return "D"  
Otherwise: return "F"

1. Finally, test your function! Call your get\_letter\_grade function with the result of get\_average(lloyd). Print the resulting letter grade.

**Script.py**

lloyd = {

"name": "Lloyd",

"homework": [90.0, 97.0, 75.0, 92.0],

"quizzes": [88.0, 40.0, 94.0],

"tests": [75.0, 90.0]

}

alice = {

"name": "Alice",

"homework": [100.0, 92.0, 98.0, 100.0],

"quizzes": [82.0, 83.0, 91.0],

"tests": [89.0, 97.0]

}

tyler = {

"name": "Tyler",

"homework": [0.0, 87.0, 75.0, 22.0],

"quizzes": [0.0, 75.0, 78.0],

"tests": [100.0, 100.0]

}

# Add your function below!

def average(numbers):

total = sum(numbers)

total = float(total)

total = total / len(numbers)

return total

def get\_average(student):

homework = average(student["homework"])

quizzes = average(student["quizzes"])

tests = average(student["tests"])

return 0.1 \* homework + 0.3 \* quizzes + 0.6 \* tests

def get\_letter\_grade(score):

if score >= 90:

return "A"

elif score >= 80:

return "B"

elif score >= 70:

return "C"

elif score >= 60:

return "D"

else:

return "F"

get\_letter\_grade(lloyd)

**Output:**

None

**Part of the Whole**

Good! Now let's calculate the class average.

You need to get the average for each student and then calculate the average of those averages.

Instructions

1. Define a function called get\_class\_average that has one argument students. You can expect students to be a list containing your three students.
2. First, make an empty list called results.
3. For each student item in the class list, calculate get\_average(student) and then call results.append() with that result.
4. Finally, return the result of calling average() with results.

**Script.py**

lloyd = {

"name": "Lloyd",

"homework": [90.0, 97.0, 75.0, 92.0],

"quizzes": [88.0, 40.0, 94.0],

"tests": [75.0, 90.0]

}

alice = {

"name": "Alice",

"homework": [100.0, 92.0, 98.0, 100.0],

"quizzes": [82.0, 83.0, 91.0],

"tests": [89.0, 97.0]

}

tyler = {

"name": "Tyler",

"homework": [0.0, 87.0, 75.0, 22.0],

"quizzes": [0.0, 75.0, 78.0],

"tests": [100.0, 100.0]

}

students = [lloyd, alice, tyler]

# Add your function below!

def average(numbers):

total = sum(numbers)

total = float(total)

total = total / len(numbers)

return total

def get\_average(student):

homework = average(student["homework"])

quizzes = average(student["quizzes"])

tests = average(student["tests"])

return 0.1 \* homework + 0.3 \* quizzes + 0.6 \* tests

def get\_letter\_grade(score):

if score >= 90:

return "A"

elif score >= 80:

return "B"

elif score >= 70:

return "C"

elif score >= 60:

return "D"

else:

return "F"

print get\_letter\_grade(lloyd)

def get\_class\_average(students):

results = []

for student in students:

results.append(get\_average(student))

return average(results)

print get\_class\_average(students)

print get\_letter\_grade(get\_class\_average(students))

**Output:**

A

83.8666666667

B

None

**Lists and Functions**

**List accessing**

This exercise goes over just pulling information from a list, which we've covered in a previous section!

Instructions

Please add the code to print out the second element in the list.

**Script.py**

n = [1, 3, 5]

# Add your code below

print n[1]

**Output:**

None

**List element modification**

You've already learned how to modify elements of a list in a previous section. This exercise is just a recap of that!

Instructions

1. On [line 3](javascript:void(0)), multiply the second element of the n list by 5
2. Overwrite the second element with that result.

Make sure to print the list when you are done!

**Script.py**

n = [1, 3, 5]

# Do your multiplication here

x = n[1] \* 5

n[1] = x

print n

**Output:**

[1, 15, 5]

None

**Appending to a list**

Here, we'll quickly recap how to .append() elements to the end of a list.

Instructions

Append the number 4 to the end of the list n.

**Script.py**

n = [1, 3, 5]

# Append the number 4 here

n.append(4)

print n

**Output:**

[1, 3, 5, 4]

None

**Removing elements from lists**

This exercise will expand on ways to remove items from a list. You actually have a few options. For a list called n:

1. n.pop(index) will remove the item at index from the list and return it to you:

n = [1, 3, 5]

n.pop(1)

# Returns 3 (the item at index 1)

print n

# prints [1, 5]

1. n.remove(item) will remove the actual item if it finds it:

n.remove(1)

# Removes 1 from the list,

# NOT the item at index 1

print n

# prints [3, 5]

1. del(n[1]) is like .pop in that it will remove the item at the given index, but it won't return it:

del(n[1])

# Doesn't return anything

print n

# prints [1, 5]

Instructions

Remove the first item from the list n using either .pop(), .remove(), or del.

**Script.py**

n = [1, 3, 5]

# Remove the first item in the list here

n.pop(0)

# or n.remove(0)

# or del(n[0])

print n

**Output:**

[3, 5]

None

**Changing the functionality of a function**

In this exercise, you will just be making a minor change to a function to change what that function does.

Instructions

Change the function so the given argument is multiplied by 3 and returned.

**Script.py**

number = 5

def my\_function(x):

return x \* 3

print my\_function(number)

**Output:**

15

None

**More than one argument**

This exercise is to recap how to use more than one argument in a function.

Instructions

Define a function called add\_function that has 2 parameters x and y and adds them together.

**Script.py**

m = 5

n = 13

# Add add\_function here!

def add\_function(x, y):

i = x + y

return i

print add\_function(m, n)

**Output:**

18

None

**Strings in functions**

This is a basic recap on using strings in functions.

Instructions

Write a function called string\_function that takes in a string argument (s) and then returns that argument concatenated with the word 'world'. **Don't** add a space before world!

**Script.py**

n = "Hello"

# Your function here!

def string\_function(s):

return s + 'world'

print string\_function(n)

**Output:**

Helloworld

None

**Passing a list to a function**

You pass a list to a function the same way you pass any other argument to a function.

Instructions

Click Save & Submit Code to see that using a list as an argument in a function is essentially the same as using just a number or string!

**Script.py**

def list\_function(x):

return x

n = [3, 5, 7]

print list\_function(n)

**Output:**

[3, 5, 7]

None

**Using an element from a list in a function**

Passing a list to a function will store it in the argument (just like with a string or a number!)

def first\_item(items):

print items[0]

numbers = [2, 7, 9]

first\_item(numbers)

1. In the example above, we define a function called first\_item. It has one argument called items.
2. Inside the function, we print out the item stored at index zero of items.
3. After the function, we create a new list called numbers.
4. Finally, we call the first\_item function with numbers as its argument, which prints out 2.

Instructions

Change [line 2](javascript:void(0)) so that list\_function returns only the item stored in index one of x, rather than the entire x list.

**Script.py**

def list\_function(x):

return x[1]

n = [3, 5, 7]

print list\_function(n)

**Output:**

5

None

**Modifying an element of a list in a function**

Modifying an element in a list in a function is the same as if you were just modifying an element of a list outside a function.

def double\_first(n):

n[0] = n[0] \* 2

numbers = [1, 2, 3, 4]

double\_first(numbers)

print numbers

1. We create a list called numbers.
2. We use the double\_first function to modify that list.
3. Finally, we print out [2, 2, 3, 4]

When we pass a list to a function and modify that list, like in the double\_first function above, we end up modifying the original list.

Instructions

Change list\_function so that:

1. Add 3 to the item at index one of the list.
2. Store the result back into index one.
3. Return the list.

**Script.py**

def list\_function(x):

x[1] = x[1] + 3

return x

n = [3, 5, 7]

print list\_function(n)

**Output:**

[3, 8, 7]

None

**List manipulation in functions**

You can also append or delete items of a list inside a function just as if you were manipulating the list outside a function.

my\_list = [1, 2, 3]

my\_list.append(4)

print my\_list

# prints [1, 2, 3, 4]

The example above is just a reminder of how to append items to a list.

Instructions

1. Define a function called list\_extender that has one parameter lst.
2. Inside the function, append the number 9 to lst.
3. Then return the modified list.

**Script.py**

n = [3, 5, 7]

# Add your function here

def list\_extender(lst):

lst.append(9)

return lst

print list\_extender(n)

**Output:**

[3, 5, 7, 9]

None

**Printing out a list item by item in a function**

This exercise is to go over how to utilize every element in a list in a function. You can use the existing code to complete the exercise and see how running this operation inside a function isn't much different from running this operation outside a function.

Don't worry about the range function quite yet—we'll explain it later in this section.

Instructions

1. Define a function called print\_list that has one argument called x.
2. Inside that function, print out each element one by one. Use the existing code as a scaffold.
3. Then call your function with the argument n.

**Script.py**

n = [3, 5, 7]

def print\_list(x):

for i in range(0, len(x)):

print x[i]

print\_list(n)

**Output:**

3

5

7

None

**Modifying each element in a list in a function**

This exercise shows how to modify each element in a list. It is useful to do so in a function as you can easily put in a list of any length and get the same functionality. As you can see, len(n) is the length of the list.

Instructions

Create a function called double\_list that takes a single argument x (which will be a list) and multiplies each element by 2 and returns that list. Use the existing code as a scaffold.

**Script.py**

n = [3, 5, 7]

def double\_list(x):

for i in range(0, len(x)):

x[i] = x[i] \* 2

return x

# Don't forget to return your new list!

print double\_list(n)

**Output:**

[6, 10, 14]

None

**Passing a range into a function**

Okay! Range time. The Python range() function is just a shortcut for generating a list, so you can use ranges in all the same places you can use lists.

range(6) # => [0,1,2,3,4,5]

range(1,6) # => [1,2,3,4,5]

range(1,6,3) # => [1,4]

The range function has three different versions:

1. **range**(stop)
2. **range**(start, stop)
3. **range**(start, stop, step)

In all cases, the range() function returns a list of numbers from start up to (but not including) stop. Each item increases by step.

If omitted, start defaults to zero and step defaults to one.

Instructions

On [line 6](javascript:void(0)), replace the \_\_\_\_ with a range() that returns a list containing [0, 1, 2].

**Script.py**

def my\_function(x):

for i in range(0, len(x)):

x[i] = x[i] \* 2

return x

print my\_function(range(3)) # Add your range between the parentheses!

**Output:**

[0, 2, 4]

None

**Iterating over a list in a function**

Now that we've learned about range, we have two ways of iterating through a list.

**Method 1** - for item in list:

for item in list:

print item

**Method 2** - iterate through indexes:

for i in range(len(list)):

print list[i]

**Method 1** is useful to loop through the list, but it's not possible to modify the list this way. **Method 2** uses indexes to loop through the list, making it possible to also modify the list if needed. Since we aren't modifying the list, feel free to use either one on this lesson!

Instructions

Create a function that returns the sum of a list of numbers.

1. On [line 3](javascript:void(0)), define a function called total that accepts one argument called numbers. It will be a list.
2. Inside the function, create a variable called result and set it to zero.
3. Using one of the two methods above, iterate through the numbers list.
4. For each number, add it to result.
5. Finally, return result.

Create a function called total that adds up all the elements of an arbitrary list and returns that count, using the existing code as a hint. Use a for loop so it can be used for any size list.

**Script.py**

n = [3, 5, 7]

def total(numbers):

result = 0

for i in range(len(numbers)):

result = result + numbers[i]

return result

print total(n)

**Output:**

15

None

**Using strings in lists in functions**

Now let's try working with strings!

for item in list:

print item

for i in range(len(list)):

print list[i]

The example above is just a reminder of the two methods for iterating over a list.

Instructions

Create a function that concatenates strings.

1. Define a function called join\_strings accepts an argument called words. It will be a list.
2. Inside the function, create a variable called result and set it to "", an empty string.
3. Iterate through the words list and append each word to result.
4. Finally, return the result.

**Don't** add spaces between the joined strings!

**Script.py**

n = ["Michael", "Lieberman"]

# Add your function here

def join\_strings(words):

result = ""

for w in range(len(words)):

result += words[w]

return result

print join\_strings(n)

**Output:**

MichaelLieberman

None

**Using two lists as two arguments in a function**

Using multiple lists in a function is no different from just using multiple arguments in a function!

a = [1, 2, 3]

b = [4, 5, 6]

print a + b

# prints [1, 2, 3, 4, 5, 6]

The example above is just a reminder of how to concatenate two lists.

Instructions

Create a function that joins two lists together.

1. On [line 4](javascript:void(0)), define a function called join\_lists that has two arguments, x and y. They will both be lists.
2. Inside that function, return the result of concatenating x and y together.

**Script.py**

m = [1, 2, 3]

n = [4, 5, 6]

# Add your code here!

def join\_lists(x, y):

return x + y

print join\_lists(m, n)

# You want this to print [1, 2, 3, 4, 5, 6]

**Output:**

[1, 2, 3, 4, 5, 6]

None

**Using a list of lists in a function**

Finally, this exercise shows how to make use of a single list that contains multiple lists and how to use them in a function.

list\_of\_lists = [[1,2,3], [4,5,6]]

for lst in list\_of\_lists:

for item in lst:

print item

1. In the example above, we first create a list containing two items, each of which is a list of numbers.
2. Then, we iterate through our outer list.
3. For each of the two inner lists (as lst), we iterate through the numbers (as item) and print them out.

We end up printing out:

1

2

3

4

5

6

Instructions

Create a function called flatten that takes a single list and concatenates all the sublists that are part of it into a single list.

1. On [line 3](javascript:void(0)), define a function called flatten with one argument called lists.
2. Make a new, empty list called results.
3. Iterate through lists. Call the looping variable numbers.
4. Iterate through numbers.
5. For each number, .append() it to results.
6. Finally, return results from your function.

**Script.py**

n = [[1, 2, 3], [4, 5, 6, 7, 8, 9]]

# Add your function here

def flatten(lists):

results = []

for lst in lists:

for item in lst:

results.append(item)

return results

print flatten(n)

**Output:**

[1, 2, 3, 4, 5, 6, 7, 8, 9]

None

**Battleship!**

**Welcome to Battleship!**

In this project you will build a simplified, one-player version of the classic board game Battleship! In this version of the game, there will be a single ship hidden in a random location on a 5x5 grid. The player will have 10 guesses to try to sink the ship.

To build this game we will use our knowledge of lists, conditionals and functions in Python. When you're ready to get started, click run to continue.

**Getting Our Feet Wet**

The first thing we need to do is to set up the game board.

Instructions

Create a variable board and set it equal to an empty list.

**Script.py**

board = []

**Output:**

None

**Make a List**

Good! Now we'll use a built-in Python function to generate our board, which we'll make into a 5 x 5 grid of all "O"s, for "ocean."

print ["O"] \* 5

will print out ['O', 'O', 'O', 'O', 'O'], which is the basis for a row of our board.

We'll do this five times to make five rows. (Since we have to do this five times, it sounds like a loop might be in order.)

Instructions

Create a 5 x 5 grid initialized to all 'O's and store it in board.

1. Use range() to loop 5 times.
2. Inside the loop, .append() a list containing 5 "O"s to board, just like in the example above.

Note that these are capital letter "O" and not zeros.

**Script.py**

board = []

for o in range(5):

board.append(5\*["O"])

print board

**Output:**

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

None

**Check it Twice**

Great job! Now that we've built our board, let's show it off.

Throughout our game, we'll want to print the game board so that the player can see which locations they have already guessed. Regularly printing the board will also help us debug our program.

The easiest way to print the board would be to have Python display it for us using the print command. Let's give that a try and see what the results look like—is this a useful way to print our board for Battleship?

Instructions

Use the print command to display the contents of the board list.

**Script.py**

board = []

for o in range(5):

board.append(5\*["O"])

print board

**Output:**

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

None

**Custom Print**

Now we can see the contents of our list, but clearly it would be easier to play the game if we could print the board out like a grid with each row on its own line.

We can use the fact that our board is a list of lists to help us do this. Let's set up a for loop to go through each of the elements in the outer list (each of which is a row of our board) and print them.

Instructions

1. First, delete your existing print statement.
2. Then, define a function named print\_board with a single argument, board.
3. Inside the function, write a for loop to iterates through each row in board and print it to the screen.
4. Call your function with board to make sure it works.

**Script.py**

board = []

for o in range(5):

board.append(5\*["O"])

def print\_board(board):

for row in board:

print row

print\_board(board)

**Output:**

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

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

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

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

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

None

**Printing Pretty**

We're getting pretty close to a playable board, but wouldn't it be nice to get rid of those quote marks and commas? We're storing our data as a list, but the player doesn't need to know that!

letters = ['a', 'b', 'c', 'd']

print " ".join(letters)

print "---".join(letters)

1. In the example above, we create a list called letters.
2. Then, we print a b c d. The .join method uses the string to combine the items in the list.
3. Finally, we print a---b---c---d. We are calling the .join function on the "---" string.

We want to turn each row into "O O O O O".

Instructions

Inside your function, inside your for loop, use " " as the separator to .join the elements of each row.

**Script.py**

board = []

for o in range(5):

board.append(5\*["O"])

def print\_board(board):

for row in board:

print " ".join(row)

print\_board(board)

**Output:**

O O O O O

O O O O O

O O O O O

O O O O O

O O O O O

None

**Hide...**

Excellent! Now, let's hide our battleship in a random location on the board.

Since we have a 2-dimensional list, we'll use two variables to store the ship's location, ship\_row and ship\_col.

from random import randint

coin = randint(0, 1)

dice = randint(1, 6)

1. In the above example, we first import the randint(low, high) function from the random module.
2. Then, we generate either zero or one and store it in coin.
3. Finally, we generate a number from one to six inclusive.

Let's generate a random\_row and random\_col from zero to four!

Instructions

1. Define two new functions, random\_row and random\_col, that each take board as input.
2. These functions should return a random row index and a random column index from your board, respectively. Use randint(0, len(board) - 1).
3. Call each function on board.

**Script.py**

from random import randint

board = []

for x in range(0, 5):

board.append(["O"] \* 5)

def print\_board(board):

for row in board:

print " ".join(row)

# Add your code below!

def random\_row(board):

return randint(0, len(board) - 1)

def random\_col(board):

return randint(0, len(board) - 1)

**Output:**

None

**...and Seek!**

Good job! For now, let's store coordinates for the ship in the variables ship\_row and ship\_col. Now you have a hidden battleship in your ocean! Let's write the code to allow the player to guess where it is.

number = raw\_input("Enter a number: ")

if int(number) == 0:

print "You entered 0"

raw\_input asks the user for input and returns it as a string. But we're going to want to use integers for our guesses! To do this, we'll wrap the raw\_inputs with int() to convert the string to an integer.

Instructions

1. Create a new variable called guess\_row and set it to int(raw\_input("Guess Row: ")).
2. Create a new variable called guess\_col and set it to int(raw\_input("Guess Col: ")).

Click **Save & Submit** and then answer the prompts by typing in a number and pressing Enter (or Return on some computers).

**Script.py**

from random import randint

board = []

for x in range(0,5):

board.append(["O"] \* 5)

def print\_board(board):

for row in board:

print " ".join(row)

def random\_row(board):

return randint(0, len(board) - 1)

def random\_col(board):

return randint(0, len(board[0]) - 1)

ship\_row = random\_row(board)

ship\_col = random\_col(board)

# Add your code below!

guess\_row = int(raw\_input("Guess Row: "))

guess\_col = int(raw\_input("Guess Col: "))

**Output:**

Guess Row: 1

Guess Col: 4

None

**It's Not Cheating—It's Debugging!**

Awesome! Now we have a hidden battleship and a guess from our player. In the next few steps, we'll check the user's guess to see if they are correct.

While we're writing and debugging this part of the program, it will be helpful to know where that battleship is hidden. Let's add a print statement that displays the location of the hidden ship.

Of course, we'll remove this output when we're finished debugging since if we left it in, our game wouldn't be very challenging. :)

Instructions

1. Print the value of ship\_col.
2. Print the value of ship\_row.

**Script.py**

from random import randint

board = []

for x in range(0,5):

board.append(["O"] \* 5)

def print\_board(board):

for row in board:

print " ".join(row)

def random\_row(board):

return randint(0, len(board) - 1)

def random\_col(board):

return randint(0, len(board[0]) - 1)

ship\_row = random\_row(board)

ship\_col = random\_col(board)

print ship\_col

print ship\_row

# Add your code below!

guess\_row = int(raw\_input("Guess Row: "))

guess\_col = int(raw\_input("Guess Col: "))

**Output:**

Guess Row: 4

Guess Col: 2

3

1

None

**You win!**

Okay—now for the fun! We have the actual location of the ship and the player's guess so we can check to see if the player guessed right.

For a guess to be right, guess\_col should be equal to ship\_col and guess\_row should be equal to ship\_row.

if guess\_col == 0 and guess\_row == 0:

print "Top-left corner."

The example above is just a reminder about if statements.

Instructions

1. On [line 29](javascript:void(0)), add an if guess\_row equals ship\_row and guess\_col equals ship\_col.
2. If that is the case, please print out "Congratulations! You sank my battleship!"

**Script.py**

from random import randint

board = []

for x in range(0, 5):

board.append(["O"] \* 5)

def print\_board(board):

for row in board:

print " ".join(row)

print\_board(board)

def random\_row(board):

return randint(0, len(board) - 1)

def random\_col(board):

return randint(0, len(board[0]) - 1)

ship\_row = random\_row(board)

ship\_col = random\_col(board)

guess\_row = int(raw\_input("Guess Row:"))

guess\_col = int(raw\_input("Guess Col:"))

# print ship\_row

# print ship\_col

# Write your code below!

if guess\_row == ship\_row and guess\_col == ship\_col:

print "Congratulations! You sank my battleship!"

**Output:**

O O O O O

O O O O O

O O O O O

O O O O O

O O O O O

4

4

Guess Row: 4

Guess Col: 4

Congratulations! You sank my battleship!

**Danger, Will Robinson!!**

Great! Of course, the player isn't going to guess right all the time, so we also need to handle the case where the guess is wrong.

print board[2][3]

The example above prints out "O", the element in the 3rd row and 4th column.

Instructions

1. Add an else under the if we wrote in the previous step.
2. Print out "You missed my battleship!"
3. Set the list element at guess\_row, guess\_col to "X".
4. As the last line in your else statement, call print\_board(board) again so you can see the "X".

Make sure to enter a col and row that is on the board!

**Script.py**

from random import randint

board = []

for x in range(0, 5):

board.append(["O"] \* 5)

def print\_board(board):

for row in board:

print " ".join(row)

print\_board(board)

def random\_row(board):

return randint(0, len(board) - 1)

def random\_col(board):

return randint(0, len(board[0]) - 1)

ship\_row = random\_row(board)

ship\_col = random\_col(board)

print ship\_row

print ship\_col

guess\_row = int(raw\_input("Guess Row:"))

guess\_col = int(raw\_input("Guess Col:"))

# Write your code below!

if guess\_row == ship\_row and guess\_col == ship\_col:

print "Congratulations! You sank my battleship!"

else:

print "You missed my battleship!"

board[guess\_row][guess\_col] = "X"

print\_board(board)

**Output:**

O O O O O

O O O O O

O O O O O

O O O O O

O O O O O

3

0

Guess Row: 2

Guess Col: 4

You missed my battleship!

O O O O O

O O O O O

O O O O X

O O O O O

O O O O O

**Bad Aim**

Great job! Now we can handle both correct and incorrect guesses from the user. But now let’s think a little bit more about the "miss" condition.

1. They can enter a guess that's off the board.
2. They can guess a spot they’ve already guessed.
3. They can just miss the ship.

We'll add these tests inside our else condition. Let's build the first case now!

if x not in range(8) or \

y not in range(3):

print "Outside the range"

The example above checks if either x or y are outside those ranges. The \ character just continues the if statement onto the next line.

Instructions

1. Add a new if: statement that is nested under the else.
2. Like the example above, it should check if guess\_row is not in range(5) or guess\_col is not in range(5).
3. If that is the case, print out "Oops, that's not even in the ocean."
4. After your new if: statement, add an else: that contains your existing handler for an incorrect guess. Don't forget to indent the code!

**Script.py**

from random import randint

board = []

for x in range(0, 5):

board.append(["O"] \* 5)

def print\_board(board):

for row in board:

print " ".join(row)

print\_board(board)

def random\_row(board):

return randint(0, len(board) - 1)

def random\_col(board):

return randint(0, len(board[0]) - 1)

ship\_row = random\_row(board)

ship\_col = random\_col(board)

print ship\_row

print ship\_col

guess\_row = int(raw\_input("Guess Row:"))

guess\_col = int(raw\_input("Guess Col:"))

# Write your code below!

if guess\_row == ship\_row and guess\_col == ship\_col:

print "Congratulations! You sank my battleship!"

else:

if guess\_row not in range(5) or guess\_col not in range(5):

print "Oops, that's not even in the ocean."

else:

board[guess\_row][guess\_col] = "X"

print "You missed my battleship!"

print\_board(board)

**Output:**

O O O O O

O O O O O

O O O O O

O O O O O

O O O O O

4

1

Guess Row: 6

Guess Col: 7

Oops, that's not even in the ocean.

O O O O O

O O O O O

O O O O O

O O O O O

O O O O O

**Not Again!**

Great! Now let's handle the second type of incorrect guess: the player guesses a location that was already guessed. How will we know that a location was previously guessed?

print board[guess\_row][guess\_col]

The example above will print an 'X' if already guessed or an 'O' otherwise.

Instructions

1. Add an elif to see if the guessed location already has an 'X' in it.
2. If it has, print "You guessed that one already."

**Script.py**

from random import randint

board = []

for x in range(0, 5):

board.append(["O"] \* 5)

def print\_board(board):

for row in board:

print " ".join(row)

print\_board(board)

def random\_row(board):

return randint(0, len(board) - 1)

def random\_col(board):

return randint(0, len(board[0]) - 1)

ship\_row = random\_row(board)

ship\_col = random\_col(board)

# print ship\_row

# print ship\_col

guess\_row = int(raw\_input("Guess Row:"))

guess\_col = int(raw\_input("Guess Col:"))

# Write your code below!

if guess\_row == ship\_row and guess\_col == ship\_col:

print "Congratulations! You sank my battleship!"

elif board[guess\_row][guess\_col] == "X":

print "You guessed that one already."

else:

if guess\_row not in range(5) or guess\_col not in range(5):

print "Oops, that's not even in the ocean."

else:

board[guess\_row][guess\_col] = "X"

print "You missed my battleship!"

print\_board(board)

**Output:**

O O O O O

O O O O O

O O O O O

O O O O O

O O O O O

Guess Row: 1

Guess Col: 2

You missed my battleship!

O O O O O

O O X O O

O O O O O

O O O O O

O O O O O

**Test Run**

Congratulations! Now you should have a game of Battleship! that is fully functional for one guess.

Make sure you play it a couple of times and try different kinds of incorrect guesses. This is a great time to stop and do some serious debugging.

In the next step, we'll move on and look at how to give the user 4 guesses to find the battleship.

Instructions

Thoroughly test your game. Make sure you try a variety of different guesses and look for any errors in the syntax or logic of your program.

**Play It, Sam**

You can successfully make one guess in Battleship! But we’d like our game to allow the player to make up to 4 guesses before they lose.

for turn in range(4):

# Make a guess

# Test that guess

We can use a for loop to iterate through a range. Each iteration will be a turn.

Instructions

1. Add a for loop that repeats the guessing and checking part of your game for 4 turns, like the example above.
2. At the beginning of each iteration, print "Turn", turn + 1 to let the player know what turn they are on.
3. Indent everything that should be repeated.

**Script.py**

from random import randint

board = []

for x in range(5):

board.append(["O"] \* 5)

def print\_board(board):

for row in board:

print " ".join(row)

print "Let's play Battleship!"

print\_board(board)

def random\_row(board):

return randint(0, len(board) - 1)

def random\_col(board):

return randint(0, len(board[0]) - 1)

ship\_row = random\_row(board)

ship\_col = random\_col(board)

print ship\_row

print ship\_col

# Everything from here on should go in your for loop!

# Be sure to indent four spaces!

for turn in range(4):

guess\_row = int(raw\_input("Guess Row:"))

guess\_col = int(raw\_input("Guess Col:"))

if guess\_row == ship\_row and guess\_col == ship\_col:

print "Congratulations! You sunk my battleship!"

else:

if (guess\_row < 0 or guess\_row > 4) or (guess\_col < 0 or guess\_col > 4):

print "Oops, that's not even in the ocean."

elif(board[guess\_row][guess\_col] == "X"):

print "You guessed that one already."

else:

print "You missed my battleship!"

board[guess\_row][guess\_col] = "X"

# Print (turn + 1) here!

print "Turn", turn + 1

print\_board(board)

**Output:**

Let's play Battleship!

O O O O O

O O O O O

O O O O O

O O O O O

O O O O O2

2

0

4

Guess Row: 2

Guess Col: 1

You missed my battleship!

Turn 1

O O O O O

O O O O O

O X O O O

O O O O O

O O O O O

Guess Row:

**Game Over**

If someone runs out of guesses without winning right now, the game just exits. It would be nice to let them know why.

Since we only want this message to display if the user guesses wrong on their last turn, we need to think carefully about where to put it.

1. We’ll want to put it under the else that accounts for misses
2. We’ll want to print the message no matter what the cause of the miss
3. Since our turn variable starts at 0 and goes to 3, we will want to end the game when turn equals 3.

Instructions

Add an if statement that checks to see if the user is out of guesses.

* Put it under the else that accounts for misses.
* Put it after the if/elif/else statements that check for the reason the player missed. We want "Game Over" to print regardless of the reason.

If turn equals 3, print "Game Over".

**Script.py**

from random import randint

board = []

for x in range(5):

board.append(["O"] \* 5)

def print\_board(board):

for row in board:

print " ".join(row)

print "Let's play Battleship!"

print\_board(board)

def random\_row(board):

return randint(0, len(board) - 1)

def random\_col(board):

return randint(0, len(board[0]) - 1)

ship\_row = random\_row(board)

ship\_col = random\_col(board)

print ship\_row

print ship\_col

# Everything from here on should go in your for loop!

# Be sure to indent four spaces!

for turn in range(4):

guess\_row = int(raw\_input("Guess Row:"))

guess\_col = int(raw\_input("Guess Col:"))

if guess\_row == ship\_row and guess\_col == ship\_col:

print "Congratulations! You sunk my battleship!"

else:

if (guess\_row < 0 or guess\_row > 4) or (guess\_col < 0 or guess\_col > 4):

print "Oops, that's not even in the ocean."

elif(board[guess\_row][guess\_col] == "X"):

print "You guessed that one already."

else:

print "You missed my battleship!"

board[guess\_row][guess\_col] = "X"

print "Turn", turn + 1

if turn == 3:

print "Game Over"

print\_board(board)

**Output:**

Let's play Battleship!

O O O O O

O O O O O

O O O O O

O O O O O

O O O O O

0

2

Guess Row: 1

Guess Col: 3

You missed my battleship!

Turn 1

O O O O O

O O O X O

O O O O O

O O O O O

O O O O O

Guess Row: 4

Guess Col: 4

You missed my battleship!

Turn 2

O O O O O

O O O X O

O O O O O

O O O O O

O O O O X

Guess Row: 4

Guess Col: 2

You missed my battleship!

Turn 3

O O O O O

O O O X O

O O O O O

O O O O O

O O X O X

Guess Row: 3

Guess Col: 1

You missed my battleship!

Turn 4

Game Over

O O O O O

O O O X O

O O O O O

O X O O O

O O X O X

**A Real Win**

Almost there! We can play Battleship!, but you’ll notice that when you win, if you haven’t already guessed 4 times, the program asks you to enter another guess. What we’d rather have happen is for the program to end—it’s no fun guessing if you know you’ve already sunk the Battleship!

We can use the command break to get out of a for loop.

Instructions

Add a break under the win condition to end the loop after a win.

**Script.py**

from random import randint

board = []

for x in range(5):

board.append(["O"] \* 5)

def print\_board(board):

for row in board:

print " ".join(row)

print "Let's play Battleship!"

print\_board(board)

def random\_row(board):

return randint(0, len(board) - 1)

def random\_col(board):

return randint(0, len(board[0]) - 1)

ship\_row = random\_row(board)

ship\_col = random\_col(board)

print ship\_row

print ship\_col

# Everything from here on should go in your for loop!

# Be sure to indent four spaces!

for turn in range(4):

guess\_row = int(raw\_input("Guess Row:"))

guess\_col = int(raw\_input("Guess Col:"))

if guess\_row == ship\_row and guess\_col == ship\_col:

print "Congratulations! You sunk my battleship!"

break

else:

if (guess\_row < 0 or guess\_row > 4) or (guess\_col < 0 or guess\_col > 4):

print "Oops, that's not even in the ocean."

elif(board[guess\_row][guess\_col] == "X"):

print "You guessed that one already."

else:

print "You missed my battleship!"

board[guess\_row][guess\_col] = "X"

print "Turn", turn + 1

if turn == 3:

print "Game Over"

print\_board(board)

**Output:**

Let's play Battleship!

O O O O O

O O O O O

O O O O O

O O O O O

O O O O O

2

3

Guess Row: 1

Guess Col: 2

You missed my battleship!

Turn 1

O O O O O

O O X O O

O O O O O

O O O O O

O O O O O

Guess Row: 2

Guess Col: 3

Congratulations! You sunk my battleship!

**To Your Battle Stations!**

Congratulations! You have a fully functional Battleship game! Play it a couple of times and get your friends to try it out, too. (Don’t forget to go back and remove the debugging output that gives away the location of the battleship!)

You may want to take some time to clean up and document your code as well.

**Extra Credit**

You can also add on to your Battleship! program to make it more complex and fun to play. Here are some ideas for enhancements—maybe you can think of some more!

1. Make multiple battleships: you'll need to be careful because you need to make sure that you don’t place battleships on top of each other on the game board. You'll also want to make sure that you balance the size of the board with the number of ships so the game is still challenging and fun to play.
2. Make battleships of different sizes: this is trickier than it sounds. All the parts of the battleship need to be vertically or horizontally touching and you’ll need to make sure you don’t accidentally place part of a ship off the side of the board.
3. Make your game a two-player game.
4. Use functions to allow your game to have more features like rematches, statistics and more!

Some of these options will be easier after we cover loops in the next lesson. Think about coming back to Battleship! after a few more lessons and see what other changes you can make!

**Loops**

**While you're here**

The while loop is similar to an if statement: it executes the code inside of it if some condition is true. The difference is that the while loop will continue to execute as long as the condition is true. In other words, instead of executing if something is true, it executes while that thing is true.

[Line 6](javascript:void(0)) decides when the loop will be executed. So, "as long as count is less than 5," the loop will continue to execute. [Line 8](javascript:void(0)) increases count by 1. This happens over and over until count equals 5.

Instructions

Change the loop so it counts up to 9 (inclusive).

Be careful not to change or remove the count += 1 bit—if Python has no way to increase count, your loop could go on forever and become an **infinite loop** which could crash your computer / browser!

**Script.py**

count = 0

if count < 5:

print "Hello, I am an if statement and count is", count

while count < 10:

print "Hello, I am a while and count is", count

count += 1

**Output:**

Hello, I am an if statement and count is 0

Hello, I am a while and count is 0

Hello, I am a while and count is 1

Hello, I am a while and count is 2

Hello, I am a while and count is 3

Hello, I am a while and count is 4

Hello, I am a while and count is 5

Hello, I am a while and count is 6

Hello, I am a while and count is 7

Hello, I am a while and count is 8

Hello, I am a while and count is 9

**Condition**

The *condition* is the expression that decides whether the loop is going to be executed or not. There are 5 steps to this program:

1. The loop\_condition variable is set to True
2. The while loop checks to see if loop\_condition is True. It is, so the loop is entered.
3. The print statement is executed.
4. The variable loop\_condition is set to False.
5. The while loop again checks to see if loop\_condition is True. It is **not**, so the loop is not executed a second time.

Instructions

See how the loop checks its condition, and when it stops executing? When you think you've got the hang of it, click Save & Submit Code to continue.

**Script.py**

loop\_condition = True

while loop\_condition:

print "I am a loop"

loop\_condition = False

**Output:**

I am a loop

**While you're at it**

Inside a while loop, you can do anything you could do elsewhere, including arithmetic operations.

Instructions

Create a while loop that prints out all the numbers from 1 to 10 squared (1, 4, 9, 16, ... , 100), each on their own line.

1. Fill in the blank space so that our while loop goes from 1 to 10 inclusive.
2. Inside the loop, print the value of num squared. The syntax for squaring a number is num \*\* 2.
3. Increment num.

**Script.py**

num = 1

while num <= 10: # Fill in the condition

# Print num squared

print num

# Increment num (make sure to do this!)

num += 1

**Output:**

1

4

9

16

25

36

49

64

81

100

**Simple errors**

A common application of a while loop is to check user input to see if it is valid. For example, if you ask the user to enter y or n and they instead enter 7, then you should re-prompt them for input.

Instructions

Fill in the loop condition so the user will be prompted for a choice over and over while choice does not equal 'y' and choice does not equal 'n'.

**Script.py**

choice = raw\_input('Enjoying the course? (y/n)')

while choice != "y" and choice != "n": # Fill in the condition (before the colon)

choice = raw\_input("Sorry, I didn't catch that. Enter again: ")

**Output:**

Enjoying the course? (y/n) 4

Sorry, I didn't catch that. Enter again: l

Sorry, I didn't catch that. Enter again: y

**Infinite loops**

An **infinite loop** is a loop that never exits. This can happen for a few reasons:

1. The loop condition cannot possibly be false (*e.g.* while 1 != 2)
2. The logic of the loop prevents the loop condition from becoming false.

Example:

count = 10

while count > 0:

count += 1 # Instead of count -= 1

Instructions

The loop in the editor has two problems: it's missing a colon (a syntax error) and count is never incremented (logical error). The latter will result in an infinite loop, so be sure to fix both before running!

**Script.py**

count = 0

while count <= 10: # Add a colon

print count

# Increment count

count += 1

**Output:**

0

1

2

3

4

5

6

7

8

9

10

**Break**

The break is a one-line statement that means "exit the current loop." An alternate way to make our counting loop exit and stop executing is with the break statement.

* First, create a while with a condition that is always true. The simplest way is shown.
* Using an if statement, you define the **stopping** condition. Inside the if, you write break, meaning "exit the loop."

The difference here is that this loop is **guaranteed** to run at least once.

Instructions

See what the break does? Feel free to mess around with it (but make sure you don't cause an infinite loop)! Click Save & Submit Code when you're ready to continue.

**Script.py**

count = 0

while True:

print count

count += 1

if count >= 10:

break

**Output:**

0

1

2

3

4

5

6

7

8

9

**While / else**

Something completely different about Python is the while/else construction. while/else is similar to if/else, but there *is* a difference: the else block will execute **anytime** the loop condition is evaluated to False. This means that it will execute if the loop is never entered or if the loop exits normally. If the loop exits as the result of a break, the else will not be executed.

In this example, the loop will break if a 5 is generated, and the else will not execute. Otherwise, after 3 numbers are generated, the loop condition will become false and the else will execute.

Instructions

Click Save & Submit Code to see while/else in action!

**Script.py**

import random

print "Lucky Numbers! 3 numbers will be generated."

print "If one of them is a '5', you lose!"

count = 0

while count < 3:

num = random.randint(1, 6)

print num

if num == 5:

print "Sorry, you lose!"

break

count += 1

else:

print "You win!"

**Output:**

Lucky Numbers! 3 numbers will be generated.

If one of them is a '5', you lose!

6

6

2

You win!

**Your own while / else**

Now you should be able to make a game similar to the one in the last exercise. The code from the last exercise is below:

count = 0

while count < 3:

num = random.randint(1, 6)

print num

if num == 5:

print "Sorry, you lose!"

break

count += 1

else:

print "You win!"

In this exercise, allow the user to guess what the number is **three** times.

guess = int(raw\_input("Your guess: "))

Remember, raw\_input turns user input into a string, so we use int() to make it a number again.

Instructions

1. Use a while loop to let the user keep guessing so long as guesses\_left is greater than zero.
2. Ask the user for their guess, just like the second example above.
3. If they guess correctly, print 'You win!' and break.
4. Decrement guesses\_left by one.
5. Use an else: case after your while loop to print You lose..

**Script.py**

from random import randint

# Generates a number from 1 through 10 inclusive

random\_number = randint(1, 10)

guesses\_left = 3

# Start your game!

while guesses\_left > 0:

guess = int(raw\_input("Your guess: "))

if guess == random\_number:

print "You win!"

break

guesses\_left -= 1

else:

print "You lose"

print random\_number

**Output:**

Your guess: 2

Your guess: 4

Your guess: 6

You lose

7

**For your health**

An alternative way to loop is the for loop. The syntax is as shown; this example means "for each number i in the range 0 - 9, print i".

Instructions

Make the loop print the numbers from 0 to 19 instead of 0 to 9.

**Script.py**

print "Counting..."

for i in range(20):

print i

**Output:**

Counting...

0

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

**For your hobbies**

This kind of loop is useful when you want to do something a certain number of times, such as append something to the end of a list.

Instructions

Create a for loop that prompts the user for a hobby 3 times, then appends each one to hobbies.

**Script.py**

hobbies = []

# Add your code below!

for hobby in range(3):

hobby = str(raw\_input("Name a hobby"))

hobbies.append(hobby)

print hobbies

**Output:**

Name a hobby art

Name a hobby fishing

Name a hobby biking

['art', 'fishing', 'biking']

**For your strings**

Using a for loop, you can print out each individual character in a string.

The example in the editor is almost plain English: "for each character c in thing, print c".

Instructions

Add a second for loop so that each character in word is printed one at a time.

**Script.py**

thing = "spam!"

for c in thing:

print c

word = "eggs!"

# Your code here!

for b in word:

print b

**Output:**

s

p

a

m

!

e

g

g

s

!

**For your A**

String manipulation is useful in for loops if you want to modify some content in a string.

word = "Marble"

for char in word:

print char,

The example above iterates through each character in word and, in the end, prints out M a r b l e.

The , character after our print statement means that our next print statement keeps printing on the same line.

Instructions

Let's filter out the letter 'A' from our string.

1. Do the following for each character in the phrase.
2. If char is an 'A' or char is an 'a', print 'X', instead of char. Make sure to include the trailing comma.
3. Otherwise (else:), please print char, with the trailing comma.

**Script.py**

phrase = "A bird in the hand..."

# Add your for loop

for char in phrase:

if char == 'A' or char == 'a':

print 'X',

else:

print char,

#Don't delete this print statement!

print

**Output:**

X b i r d i n t h e h X n d . . .

**For your lists**

Perhaps the most useful (and most common) use of for loops is to go through a list.

On each iteration, the variable num will be the next value in the list. So, the first time through, it will be 7, the second time it will be 9, then 12, 54, 99, and then the loop will exit when there are no more values in the list.

Instructions

Write a second for loop that goes through the numbers list and prints each element *squared*, each on its own line.

**Script.py**

numbers = [7, 9, 12, 54, 99]

print "This list contains: "

for num in numbers:

print num

# Add your loop below!

for num in numbers:

print num \*\*2

**Output:**

This list contains:

7

9

12

54

99

49

81

144

2916

9801

**Looping over a dictionary**

You may be wondering how looping over a dictionary would work. Would you get the key or the value?

The short answer is: you get the key which you can use to get the value.

d = {'x': 9, 'y': 10, 'z': 20}

for key in d:

if d[key] == 10:

print "This dictionary has the value 10!"

1. First, we create a dictionary with strings as the keys and numbers as the values.
2. Then, we iterate through the dictionary, each time storing the key in key.
3. Next, we check if that key's value is equal to 10.
4. Finally, we print This dictionary has the value 10!

Instructions

On [line 5](javascript:void(0)), print the key, followed by a space, followed by the value associated with that key.

**Script.py**

d = {'a': 'apple', 'b': 'berry', 'c': 'cherry'}

for key in d:

# Your code here!

print key, d[key]

**Output:**

a apple

c cherry

b berry

**Counting as you go**

A weakness of using this for-each style of iteration is that you don't know the index of the thing you're looking at. Generally this isn't an issue, but at times it is useful to know how far into the list you are. Thankfully the built-in enumerate function helps with this.

enumerate works by supplying a corresponding index to each element in the list that you pass it. Each time you go through the loop, index will be one greater, and item will be the next item in the sequence. It's very similar to using a normal for loop with a list, except this gives us an easy way to count how many items we've seen so far.

Instructions

We don't want the user to see things listed from index 0, since this looks unnatural. Instead, the items should appear to start at index 1. Modify the print statement to reflect this behavior. See the Hint for help.

**Script.py**

choices = ['pizza', 'pasta', 'salad', 'nachos']

print 'Your choices are:'

for index, item in enumerate(choices):

print index+1, item

**Output:**

Your choices are:

1 pizza

2 pasta

3 salad

4 nachos

**Multiple lists**

It's also common to need to iterate over two lists at once. This is where the built-in zip function comes in handy.

zip will create pairs of elements when passed two lists, and will stop at the end of the shorter list.

zip can handle three or more lists as well!

Instructions

Compare each pair of elements and print the larger of the two.

**Script.py**

list\_a = [3, 9, 17, 15, 19]

list\_b = [2, 4, 8, 10, 30, 40, 50, 60, 70, 80, 90]

for a, b in zip(list\_a, list\_b):

# Add your code here!

if a > b:

print a

else:

print b

**Output:**

3

9

17

15

30

**For / else**

Just like with while, for loops may have an else associated with them.

In this case, the else statement is executed after the for, but *only* if the for ends normally—that is, not with a break. This code will break when it hits 'tomato', so the else block won't be executed.

Instructions

Click Save & Submit Code to see how for and else work together.

**Script.py**

fruits = ['banana', 'apple', 'orange', 'tomato', 'pear', 'grape']

print 'You have...'

for f in fruits:

if f == 'tomato':

print 'A tomato is not a fruit!' # (It actually is.)

break

print 'A', f

else:

print 'A fine selection of fruits!'

**Output:**

You have...

A banana

A apple

A orange

A tomato is not a fruit!

**Change it up**

As mentioned, the else block won't run in this case, since break executes when it hits 'tomato'.

Instructions

Modify the code in the editor such that the for loop's else statement is executed.

**Script.py**

fruits = ['banana', 'apple', 'orange', 'tomato', 'pear', 'grape']

print 'You have...'

for f in fruits:

if f == 'pineapple':

print 'A pineapple a fruit!' # (It actually is.)

break

print 'A', f

else:

print 'A fine selection of fruits!'

**Output:**

You have...

A banana

A apple

A orange

A tomato

A pear

A grape

A fine selection of fruits!

**Create your own**

To wrap up this lesson, let's create our own for/else statement from scratch.

Instructions

Build your for/else statement in the editor. Execution of the else branch is optional, but your code should print a string of your choice to the editor regardless.

**Script.py**

heroes = ["Batman", "Superman", "Iron Man"]

for h in heroes:

if h == "Wonder Woman":

print "She's tough"

break

print h

else:

print "No girls allowed!"

**Output:**

Batman

Superman

Iron Man

No girls allowed!

**Practice Makes Perfect**

**Practice! Practice Practice!**

The best way to get good at anything is a lot of practice. This lesson is full of practice problems for you to work on. This section will contain **minimal instructions** to help you solve these problems; instead, this section will help you work on taking your programming skills and applying them to real life problems.

The more challenging programs will contain some helpful hints to nudge you in the right direction. If you feel as if you are completely lost, feel free to check out the Q&A section for help (the link is on the very bottom left of your screen).

Instructions

Hit Save & Submit Code to continue.

**is\_even**

All right! Let's get started.

Remember how an even number is a number that is divisible by 2?

Instructions

1. Define a function is\_even that will take a number x as input.
2. If x is even, then return True.
3. Otherwise, return False.

**Script.py**

def is\_even(x):

if x % 2:

return False

else:

return True

**Output:**

None

**is\_int**

An integer is just a number without a decimal part (for instance, -17, 0, and 42 are all integers, but 98.6 is not).

For the purpose of this lesson, we'll also say that **a number with a decimal part that is all 0s is also an integer**, such as 7.0.

This means that, for this lesson, you *can't* just test the input to see if it's of type int.

If the difference between a number and that same number rounded down is greater than zero, what does that say about that particular number?

Instructions

1. Define a function is\_int that takes a number x as an input.
2. Have it return True if the number is an integer (as defined above) and False otherwise.

For example:

is\_int(7.0) # True

is\_int(7.5) # False

is\_int(-1) # True

**Script.py**

def is\_int(x):

if x == int(x):

return True

else:

return False

is\_int(7.0)

**Output:**

None

**digit\_sum**

Awesome! Now let's try something a little trickier. Try summing the digits of a number.

Instructions

Write a function called digit\_sum that takes a positive integer n as input and returns the sum of all that number's digits.

For example: digit\_sum(1234) should return 10 which is 1 + 2 + 3 + 4.

(Assume that the number you are given will always be positive.)

Check the hint if you need help!

**Script.py**

def digit\_sum(n):

l = []

m = str(n)

for i in m:

j = int(i)

l.append(j)

return sum(list(l))

print digit\_sum(123456)

**Output:**

21

**factorial**

All right! Now we're cooking. Let's try a factorial problem.

To calculate the factorial of a non-negative integer x, just multiply all the integers from 1 through x. For example:

* factorial(4) would equal 4 \* 3 \* 2 \* 1, which is 24.
* factorial(1) would equal 1.
* factorial(3) would equal 3 \* 2 \* 1, which is 6.

Instructions

Define a function factorial that takes an integer x as input.

Calculate and return the factorial of that number.

**Script.py**

def factorial(x):

result = 1

if x > 0:

while x > 0:

result \*= x

x -= 1

return result

print factorial(4)

**Output:** 24

**is\_prime**

A **prime** number is a positive integer greater than 1 that has no positive divisors other than 1 and itself. (Boy, that's a mouthful.)

In other words, if you want to test if a number in a variable x is prime, then no other number should go into x evenly besides 1 and x. So 2 and 5 and 11 are all prime, but 4 and 18 and 21 are not.

If there is a number between 1 and x that goes in evenly, then x is not prime.

Instructions

1. Define a function called is\_prime that takes a number x as input.
2. For each number n from 2 to x - 1, test if x is evenly divisible by n.
3. If it is, return False.
4. If none of them are, then return True.

**Script.py**

def is\_prime(x):

if x < 2:

return False

for n in range(2,x):

if (x % n) == 0:

return False

return True

print is\_prime(21)

**Output:**

False

**reverse**

Great work so far! Let's practice writing some functions that work with strings.

Instructions

Define a function called reverse that takes a string textand returns that string in reverse.

For example: reverse("abcd") should return "dcba".

1. You may not use reversed or [::-1] to help you with this.
2. You may get a string containing special characters (for example, !, @, or #).

**Script.py**

def reverse(text):

x = len(text)-1

new = ""

while x >= 0:

new += text[x]

x = x -1

return new

print reverse("Hello")

**Output:**

olleH

**anti\_vowel**

Nice work. Next up: vowels!

Instructions

Define a function called anti\_vowel that takes one string, text, as input and returns the text with all of the vowels removed.

For example: anti\_vowel("Hey You!") should return "Hy Y!".

* Don't count Y as a vowel.
* Make sure to remove lowercase **and** uppercase vowels.

**Script.py**

def anti\_vowel(text):

for x in text:

if x in "aeiouAEIOU":

text = text.replace(x, '')

return text

print anti\_vowel("Hey You!")

**Output:**

Hy Y!

**scrabble\_score**

Scrabble is a game where players get points by spelling words. Words are scored by adding together the point values of each individual letter (we'll leave out the double and triple letter and word scores for now).

To the right is a dictionary containing all of the letters in the alphabet with their corresponding Scrabble point values.

For example: the word "Helix" would score 15 points due to the sum of the letters: 4 + 1 + 1 + 1 + 8.

Instructions

Define a function scrabble\_score that takes a string word as input and returns the equivalent scrabble score for that word.

1. Assume your input is only one word containing no spaces or punctuation.
2. As mentioned, no need to worry about score multipliers!
3. Your function should work even if the letters you get are uppercase, lowercase, or a mix.
4. Assume that you're only given non-empty strings.

**Script.py**

score = {"a": 1, "c": 3, "b": 3, "e": 1, "d": 2, "g": 2,

"f": 4, "i": 1, "h": 4, "k": 5, "j": 8, "m": 3,

"l": 1, "o": 1, "n": 1, "q": 10, "p": 3, "s": 1,

"r": 1, "u": 1, "t": 1, "w": 4, "v": 4, "y": 4,

"x": 8, "z": 10}

def scrabble\_score(word):

word = word.lower()

total = 0

for w in word:

total += score[w]

return total

print scrabble\_score("Helix")

**Output:**

15

**censor**

You're doing great with these string function challenges. Last one!

Instructions

Write a function called censor that takes two strings, text and word, as input. It should return the text with the word you chose replaced with asterisks.

For example:

censor("this hack is wack hack", "hack")

should return

"this \*\*\*\* is wack \*\*\*\*"

1. Assume your input strings won't contain punctuation or upper case letters.
2. The number of asterisks you put should correspond to the number of letters in the censored word.

**Script.py**

def censor(text, word):

censorWord = ''.join('\*' for char in word)

return text.replace(word,censorWord)

print censor("this hack is wack hack", "hack")

**Output:**

this \*\*\*\* is wack \*\*\*\*

**count**

Great work so far. Let's finish up by practicing with a few functions that take lists as arguments.

Instructions

1. Define a function called count that has two arguments called sequence and item.
2. Return the number of times the item occurs in the list.

For example: count([1,2,1,1], 1) should return 3 (because 1 appears 3 times in the list).

1. There *is* a list method in Python that you can use for this, but you should do it the long way for practice.
2. Your function should return an integer.
3. The item you input may be an integer, string, float, or even another list!
4. Be careful not to use list as a variable name in your code—it's a reserved word in Python!

**Script.py**

def count(sequence, item):

num = 0

for i in sequence:

if i == item:

num += 1

return num

print count([5, 10, 2, 5, 4, 5, 3], 5)

**Output:**

3

**purify**

Awesome

**Do not** directly modify the list you are given as input; instead, return a new list with only the even numbers.

e! Now let's practice filtering a list.

Instructions

Define a function called purify that takes in a list of numbers, removes all odd numbers in the list, and returns the result.

For example, purify([1,2,3]) should return [2].

**Script.py**

def purify(x):

new = []

for i in x:

if i % 2 == 0:

new.append(i)

return new

print purify([1,2,3,4,5,6,7,8,9,10])

**Output:**

[2, 4, 6, 8, 10]

**product**

Great! Now let's try a little multiplication.

Instructions

Define a function called product that takes a list of integers as input and returns the product of all of the elements in the list.

For example: product([4, 5, 5]) should return 100 (because 4 \* 5 \* 5 is 100).

1. Don't worry about the list being empty.
2. Your function should return an integer.

**Script.py**

def product(x):

n = 1

for i in x:

n \*= i

return n

print product([4,5,5])

**Output:**

100

**remove\_duplicates**

Awesome! Now for something a bit trickier.

Instructions

Write a function remove\_duplicates that takes in a list and removes elements of the list that are the same.

For example: remove\_duplicates([1,1,2,2])   
should return [1,2].

1. Don't remove *every* occurrence, since you need to keep a single occurrence of a number.
2. The order in which you present your output does not matter. So returning [1,2,3] is the same as returning [3,1,2].
3. **Do not** modify the list you take as input! Instead, return a new list.

**Script.py**

def remove\_duplicates(x):

new = []

for i in range(len(x)):

if x[i] not in new:

new.append(x[i])

return new

print remove\_duplicates([4,5,5,4])

**Output:**

[4,5]

**median**

Great work! You've covered a lot in these exercises. Last but not least, let's write a function to find the median of a list.

The **median** is the middle number in a sorted sequence of numbers.

Finding the median of [7,12,3,1,6] would first consist of sorting the sequence into [1,3,6,7,12] and then locating the middle value, which would be 6.

If you are given a sequence with an **even** number of elements, you must average the two elements surrounding the middle.

For example, the median of the sequence [7,3,1,4] is 3.5, since the middle elements after sorting the list are 3 and 4 and (3 + 4) / (2.0) is 3.5.

You can sort the sequence using the sorted() function, like so:

sorted([5, 2, 3, 1, 4])

[1, 2, 3, 4, 5]

Instructions

Write a function called median that takes a list as an input and returns the median value of the list.

For example: median([1,1,2]) should return 1.

1. The list can be of any size and the numbers are not guaranteed to be in any particular order.
2. If the list contains an even number of elements, your function should return the average of the middle two.

**Script.py**

def median(x):

x.sort()

l = len(x)

for num in x:

if l % 2 == 0:

a = (x[l/2] + x[l/2 - 1]) / 2.0

return a

elif l == 1:

return (x[0])

else:

a2 = x[l/2]

return a2

print median([4,5,5,4])

**Output**:

4.5

**Exam Statistics**

**Let's look at those grades!**

Creating a program to compute statistics means that you won't have to whip out your calculator and manually crunch numbers. All you'll have to do is supply a new set of numbers and our program does all of the hard work.

This mini-project will give you some practice with functions, lists, and translating mathematical formulae into programming statements.

In order to use the scores in our program, we'll need them in a container, namely a list.

On the right, you'll see the grades *listed* (see what I did there). The data is anonymous to protect the privacy of the students.

Instructions

Hit Save & Submit Code to continue.

**Script.py**

grades = [100, 100, 90, 40, 80, 100, 85, 70, 90, 65, 90, 85, 50.5]

print "Grades:", grades

**Output:**

Grades: [100, 100, 90, 40, 80, 100, 85, 70, 90, 65, 90, 85, 50.5]

**Print those grades**

As a refresher, let's start off by writing a function to print out the list of grades, **one element at a time**.

Instructions

1. Define a function on [line 3](javascript:void(0)) called print\_grades() with one argument, a list called grades.
2. Inside the function, iterate through grades and print each item on its own line.
3. After your function, call print\_grades() with the grades list as the parameter.

**Script.py**

grades = [100, 100, 90, 40, 80, 100, 85, 70, 90, 65, 90, 85, 50.5]

def print\_grades(grades):

for grade in grades:

print grade

print\_grades(grades)

**Output:**

100

100

90

40

80

100

85

70

90

65

90

85

50.5

**Review**

So far, you've created a helper function that will be used in the next sections.

You also have a solid handle on the concepts that we'll need to continue.

The next step in the creation of our grade statistics program involves computing the mean (average) of the grades.

Onwards.

Instructions

Hit Save & Submit Code to continue.

**Script.py**

print "Let's compute some stats!"

**Output:**

Let's compute some stats!

**The sum of scores**

Now that we have a function to print the grades, let's create another function to compute the sum of all of the test grades.

This will be super-helpful when we need to compute the average score.

I know what you're thinking, "let's just use the built-in sum() function!" The built-in function would work beautifully, but it would be too easy.

Computing the sum manually involves computing a rolling sum. As you loop through the list, add the current grade to a variable that keeps track of the total, let's call that variable total.

Instructions

On [line 3](javascript:void(0)), define a function grades\_sum() that does the following:

1. Takes in a list of scores, scores
2. Computes the sum of the scores
3. Returns the computed sum

Call the newly created grades\_sum() function with the list of grades and print the result.

**Script.py**

grades = [100, 100, 90, 40, 80, 100, 85, 70, 90, 65, 90, 85, 50.5]

def grades\_sum(scores):

total = 0

for i in scores:

total += i

return total

print grades\_sum(grades)

**Output:**

1045.5

**Computing the Average**

The average test grade can be found by dividing the sum of the grades by the total number of grades.

Luckily, we just created an awesome function, grades\_sum() to compute the sum.

Instructions

Define a function grades\_average(), below the grades\_sum() function that does the following:

1. Has one argument, grades, a list
2. Calls grades\_sum with grades
3. Computes the average of the grades by dividing that sum by float(len(grades)).
4. Returns the average.

Call the newly created grades\_average() function with the list of grades and **print the result**.

**Script.py**

grades = [100, 100, 90, 40, 80, 100, 85, 70, 90, 65, 90, 85, 50.5]

def grades\_average(grades):

def grades\_sum(scores):

total = 0

for scores in grades:

total += scores

return total

return grades\_sum(grades) / float(len(grades))

print grades\_sum(grades)

print grades\_average(grades)

**Output:**

1045.5

80.4230769231

**Review**

Great work creating the capability to compute the average of the test grades.

We're going to use the average for computing the variance. The variance allows us to see how widespread the grades were from the average.

**Script.py**

print "Time to conquer the variance!"

**Output:**

Time to conquer the variance!

**The Variance**

Let's see how the grades varied against the average. This is called computing the **variance**.

A very large variance means that the students' grades were all over the place, while a small variance (relatively close to the average) means that the majority of students did fairly well.

Instructions

1. On [line 18](javascript:void(0)), define a new function called grades\_variance() that accepts one argument, scores, a list.
2. First, create a variable average and store the result of calling grades\_average(scores).
3. Next, create another variable variance and set it to zero. We will use this as a rolling sum.
4. for each score in scores: Compute its squared difference: (average - score) \*\* 2 and add that to variance.
5. Divide the total variance by the number of scores.
6. Then, return that result.
7. Finally, after your function code, print grades\_variance(grades).

**Script.py**

grades = [100, 100, 90, 40, 80, 100, 85, 70, 90, 65, 90, 85, 50.5]

def print\_grades(grades):

for grade in grades:

print grade

def grades\_sum(grades):

total = 0

for grade in grades:

total += grade

return total

def grades\_average(grades):

sum\_of\_grades = grades\_sum(grades)

average = sum\_of\_grades / float(len(grades))

return average

def grades\_variance(scores):

average = grades\_average(scores)

variance = 0

for score in scores:

variance += (average - score)\*\*2

result = variance / float(len(scores))

return result

print grades\_variance(grades)

**Output:**

334.071005917

**Standard Deviation**

Great job computing the variance! The last statistic will be much simpler: standard deviation.

The standard deviation is the square root of the variance. You can calculate the square root by raising the number to the one-half power.

Instructions

1. Define a function grades\_std\_deviation(variance).
2. return the result of variance \*\* 0.5
3. After the function, create a new variable called variance and store the result of calling grades\_variance(grades).
4. Finally print the result of calling grades\_std\_deviation(variance).

**Script.py**

grades = [100, 100, 90, 40, 80, 100, 85, 70, 90, 65, 90, 85, 50.5]

def print\_grades(grades):

for grade in grades:

print grade

def grades\_sum(grades):

total = 0

for grade in grades:

total += grade

return total

def grades\_average(grades):

sum\_of\_grades = grades\_sum(grades)

average = sum\_of\_grades / float(len(grades))

return average

def grades\_variance(scores):

average = grades\_average(scores)

variance = 0

for score in scores:

score = (average - score)\*\*2

variance += score

result = variance / float(len(scores))

return result

print grades\_variance(grades)

def grades\_std\_deviation(variance):

return variance \*\* 0.5

variance = grades\_variance(grades)

print grades\_std\_deviation(variance)

**Output:**

29.4812243969

5.42966153613

**Review**

You've done a great job completing this program.

We've created quite a few meaningful functions. Namely, we've created helper functions to print a list of grades, compute the sum, average, variance, and standard deviation about a set of grades.

Let's wrap up by printing out all of the statistics.

Who needs to pay for grade calculation software when you can write your own? :)

Instructions

Print out the following:

* all of the grades
* sum of grades
* average grade
* variance
* standard deviation

**Script.py**

grades = [100, 100, 90, 40, 80, 100, 85, 70, 90, 65, 90, 85, 50.5]

def print\_grades(grades):

for grade in grades:

print grade

print print\_grades(grades)

def grades\_sum(grades):

total = 0

for grade in grades:

total += grade

return total

print grades\_sum(grades)

def grades\_average(grades):

sum\_of\_grades = grades\_sum(grades)

average = sum\_of\_grades / float(len(grades))

return average

print grades\_average(grades)

def grades\_variance(scores):

average = grades\_average(scores)

variance = 0

for score in scores:

score = (average - score)\*\*2

variance += score

result = variance / float(len(scores))

return result

print grades\_variance(grades)

def grades\_std\_deviation(variance):

return variance \*\* 0.5

variance = grades\_variance(grades)

print grades\_std\_deviation(variance)

**Output:**

100

100

90

40

80

100

85

70

90

65

90

85

50.5

None

1045.5

80.4230769231

334.071005917

18.2776094147

**Taking a Vacation**

**Before We Begin**

Let's first quickly review functions in Python.

def bigger(first, second):

print max(first, second)

return True

In the example above:

1. We define a function called bigger that has two arguments called first and second.
2. Then, we print out the larger of the two arguments using the built-in function max.
3. Finally, the bigger function returns True.

Now try creating a function yourself!

Instructions

Write a function called answer that takes no arguments and returns the value 42.

Even without arguments, you will still need parentheses.

Don't forget the colon at the end of the function definition!

**Script.py**

def answer():

x = 42

return x

OR

return 42

**Output:**

42

**Planning Your Trip**

When planning a vacation, it's very important to know exactly how much you're going to spend.

def wages(hours):

# If I make $8.35/hour...

return 8.35 \* hours

The above example is just a refresher in how functions are defined.

Let's use functions to calculate your trip's costs.

Instructions

1. Define a function called hotel\_cost with one argument nights as input.
2. The hotel costs $140 per night. So, the function hotel\_cost should return 140 \* nights.

**Script.py**

def hotel\_cost(nights):

return 140 \* nights

**Output:**

None

**Getting There**

You're going to need to take a plane ride to get to your location.

def fruit\_color(fruit):

if fruit == "apple":

return "red"

elif fruit == "banana":

return "yellow"

elif fruit == "pear":

return "green"

1. The example above defines the function fruit\_color that accepts a string as the argument fruit.
2. The function returns a string if it knows the color of that fruit.

Instructions

1. Below your existing code, define a function called plane\_ride\_cost that takes a string, city, as input.
2. The function should return a different price depending on the location, similar to the code example above. Below are the valid destinations and their corresponding round-trip prices.

"Charlotte": 183  
"Tampa": 220  
"Pittsburgh": 222  
"Los Angeles": 475

**Script.py**

def hotel\_cost(nights):

return 140 \* nights

def plane\_ride\_cost(city):

if city == "Charlotte":

return 183

elif city == "Tampa":

return 220

elif city == "Pittsburgh":

return 222

else:

return 475

**Output:**

None

**Transportation**

You're also going to need a rental car in order for you to get around.

def finish\_game(score):

tickets = 10 \* score

if score >= 10:

tickets += 50

elif score >= 7:

tickets += 20

return tickets

In the above example, we first give the player 10 tickets for every point that the player scored. Then, we check the value of score multiple times.

1. First, we check if score is greater than or equal to 10. If it is, we give the player 50 bonus tickets.
2. If score is just greater than or equal to 7, we give the player 20 bonus tickets.
3. At the end, we return the total number of tickets earned by the player.

Remember that an elif statement is only checked if all preceding if/elif statements fail.

Instructions

1. Below your existing code, define a function called rental\_car\_cost with an argument called days.
2. Calculate the cost of renting the car:
   * Every day you rent the car costs $40.
   * if you rent the car for 7 or more days, you get $50 off your total.
   * *Alternatively* (elif), if you rent the car for 3 or more days, you get $20 off your total.
   * You cannot get both of the above discounts.
3. Return that cost.

Just like in the example above, this check becomes simpler if you make the 7-day check an if statement and the 3-day check an elif statement.

**Script.py**

def hotel\_cost(nights):

return 140 \* nights

def plane\_ride\_cost(city):

if city == "Charlotte":

return 183

elif city == "Tampa":

return 220

elif city == "Pittsburgh":

return 222

else:

return 475

def rental\_car\_cost(days):

costs = 40

if days >= 7:

return days \* costs - 50

elif days >= 3:

return days \* costs - 20

else:

return days \* costs

**Output:**

None

**Pull it Together**

Great! Now that you've got your 3 main costs figured out, let's put them together in order to find the total cost of your trip.

def double(n):

return 2 \* n

def triple(p):

return 3 \* p

def add(a, b):

return double(a) + triple(b)

1. We define two simple functions, double(n) and triple(p) that return 2 times or 3 times their input. Notice that they have n and p as their **arguments**.
2. We define a third function, add(a, b) that returns the sum of the previous two functions when called with a and b, respectively.

Instructions

1. Below your existing code, define a function called trip\_cost that takes two arguments, city and days.
2. Like the example above, have your function return the **sum** of calling the rental\_car\_cost(days), hotel\_cost(days), and plane\_ride\_cost(city) functions.

It is completely valid to call the hotel\_cost(nights) function with the variable days. Just like the example above where we call double(n) with the variable a, we pass the value of days to the new function in the argument nights.

**Script.py**

def hotel\_cost(nights):

return 140 \* nights

def plane\_ride\_cost(city):

if city == "Charlotte":

return 183

elif city == "Tampa":

return 220

elif city == "Pittsburgh":

return 222

else:

return 475

def rental\_car\_cost(days):

costs = 40

if days >= 7:

return days \* costs - 50

elif days >= 3:

return days \* costs - 20

else:

return days \* costs

def trip\_cost(city, days):

d = rental\_car\_cost(days)

n = hotel\_cost(days)

c = plane\_ride\_cost(city)

return d + n + c

print trip\_cost("Tampa", 6)

**Output:**

1280

**Hey, You Never Know!**

You can't expect to only spend money on the plane ride, hotel, and rental car when going on a vacation. There also needs to be room for additional costs like fancy food or souvenirs.

Instructions

1. Modify your trip\_cost function definition. Add a third argument, spending\_money.
2. Modify what the trip\_cost function does. Add the variable spending\_money to the sum that it returns.

**Script.py**

def hotel\_cost(nights):

return 140 \* nights

def plane\_ride\_cost(city):

if city == "Charlotte":

return 183

elif city == "Tampa":

return 220

elif city == "Pittsburgh":

return 222

else:

return 475

def rental\_car\_cost(days):

costs = 40

if days >= 7:

return days \* costs - 50

elif days >= 3:

return days \* costs - 20

else:

return days \* costs

def trip\_cost(city, days, spending\_money):

d = rental\_car\_cost(days)

n = hotel\_cost(days)

c = plane\_ride\_cost(city)

m = spending\_money

return d + n + c + m

print trip\_cost("Tampa", 6, 300)

**Output:**

1580

**Plan Your Trip!**

Nice work! Now that you have it all together, let's take a trip.

What if we went to Los Angeles for 5 days and brought an extra 600 dollars of spending money?

Instructions

After your previous code, print out the trip\_cost( to "Los Angeles" for 5 days with an extra 600 dollars of spending money.

Don't forget the closing ) after passing in the 3 previous values!

**Script.py**

def hotel\_cost(nights):

return 140 \* nights

def plane\_ride\_cost(city):

if city == "Charlotte":

return 183

elif city == "Tampa":

return 220

elif city == "Pittsburgh":

return 222

else:

return 475

def rental\_car\_cost(days):

costs = 40

if days >= 7:

return days \* costs - 50

elif days >= 3:

return days \* costs - 20

else:

return days \* costs

def trip\_cost(city, days, spending\_money):

d = rental\_car\_cost(days)

n = hotel\_cost(days)

c = plane\_ride\_cost(city)

m = spending\_money

return d + n + c + m

print trip\_cost("Los Angeles", 5, 600)

**Output:**

**1955**

**A Day at the Supermarket**

**BeFOR We Begin**

Before we begin our exercise, we should go over the Python for loop one more time. For now, we are only going to go over the for loop in terms of how it relates to lists and dictionaries. We'll explain more cool for loop uses in later courses.

for loops allow us to iterate through all of the elements in a list from the left-most (or zeroth element) to the right-most element. A sample loop would be structured as follows:

a = ["List of some sort”]

for x in a:

# Do something for every x

This loop will run all of the code in the indented block under the for x in a: statement. The item in the list that is currently being evaluated will be x. So running the following:

for item in [1, 3, 21]:

print item

would print 1, then 3, and then 21. The variable between for and in can be set to any variable name (currently item), but you should be careful to avoid using the word “list” as a variable, since that's a reserved word (that is, it means something special) in the Python language.

Instructions

Use a for loop to print out all of the elements in the list names.

**Script.py**

names = ["Adam","Alex","Mariah","Martine","Columbus"]

for name in names:

print name

**Output:**

Adam

Alex

Mariah

Martine

Columbus

**This is KEY!**

You can also use a for loop on a dictionary to loop through its *keys* with the following:

# A simple dictionary

d = {"foo" : "bar"}

for key in d:

print d[key] # prints "bar"

Note that dictionaries are **unordered**, meaning that any time you loop through a dictionary, you will go through *every* key, but you are not guaranteed to get them in any particular order.

Instructions

Use a for loop to go through the webster dictionary and print out all of the definitions.

**Script.py**

webster = {

"Aardvark" : "A star of a popular children's cartoon show.",

"Baa" : "The sound a goat makes.",

"Carpet": "Goes on the floor.",

"Dab": "A small amount."

}

# Add your code below!

for key in webster:

print webster[key]

**Output:**

A star of a popular children's cartoon show.

Goes on the floor.

A small amount.

The sound a goat makes.

**Control Flow and Looping**

The blocks of code in a for loop can be as big or as small as they need to be.

While looping, you may want to perform different actions depending on the particular item in the list.

numbers = [1, 3, 4, 7]

for number in numbers:

if number > 6:

print number

print "We printed 7."

1. In the above example, we create a list with 4 numbers in it.
2. Then we loop through the numbers list and store each item in the list in the variable number.
3. On each loop, if number is greater than 6, we print it out. So, we print 7.
4. Finally, we print out a sentence.

Make sure to keep track of your indentation or you may get confused!

Instructions

1. Like step 2 above, loop through each item in the list called a.
2. Like step 3 above, if the number is even, print it out. You can test if the item % 2 == 0 to help you out.

**Script.py**

a = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]

for num in a:

if num % 2 == 0:

print num

**Output:**

0

2

4

6

8

10

12

**Lists + Functions**

Functions can also take lists as inputs and perform various operations on those lists.

def count\_small(numbers):

total = 0

for n in numbers:

if n < 10:

total = total + 1

return total

lost = [4, 8, 15, 16, 23, 42]

small = count\_small(lost)

print small

1. In the above example, we define a function count\_small that has one argument, numbers.
2. We initialize a variable total that we can use in the for loop.
3. For each item n in numbers, if n is less than 10, we increment total.
4. After the for loop, we return total.
5. After the function definition, we create an array of numbers called lost.
6. We call the count\_small function, pass in lost, and store the returned result in small.
7. Finally, we print out the returned result, which is 2 since only 4 and 8 are less than 10.

Instructions

Write a function that counts how many times the string "fizz" appears in a list.

1. Write a function called fizz\_count that takes a list x as input.
2. Create a variable count to hold the ongoing count. Initialize it to zero.
3. for each item in x:, if that item is equal to the string "fizz" then increment the count variable.
4. After the loop, please return the count variable.

For example, fizz\_count(["fizz","cat","fizz"]) should return 2.

**Script.py**

# Write your function below!

def fizz\_count(x):

count = 0

for item in x:

if item == "fizz":

count = count + 1

return count

fizz = fizz\_count(["fizz","cat","fizz"])

print fizz

**Output:** 2

**String Looping**

As we've mentioned, strings are like lists with characters as elements. You can loop through strings the same way you loop through lists! While we won't ask you to do that in this section, we've put an example in the editor of how looping through a string might work.

Instructions

Run the code to see string iteration in action!

**Script.py**

for letter in "Codecademy":

print letter

# Empty lines to make the output pretty

print

print

word = "Programming is fun!"

for letter in word:

# Only print out the letter i

if letter == "i":

print letter

**Output:**

C

o

d

e

c

a

d

e

m

y

i

i

**Your Own Store!**

Okay—on to the core of our project.

Congratulations! You are now the proud owner of your very own Codecademy brand supermarket.

animal\_counts = {

"ant": 3,

"bear": 6,

"crow": 2

}

In the example above, we create a new dictionary called animal\_counts with three entries. One of the entries has the key "ant" and the value 3.

Instructions

1. Create a new dictionary called prices using {} format like the example above.
2. Put these values in your prices dictionary, in between the {}:

"banana": 4,

"apple": 2,

"orange": 1.5,

"pear": 3

Yeah, this place is really expensive. (Your supermarket subsidizes the zoo from the last course.)

**Script.py**

prices = {

"banana": 4,

"apple": 2,

"orange": 1.5,

"pear": 3

}

**Output:**

None

**Investing in Stock**

Good work! As a store manager, you’re also in charge of keeping track of your stock/inventory.

Instructions

Create a stock dictionary with the values below.

"banana": 6

"apple": 0

"orange": 32

"pear": 15

**Script.py**

prices = {

"banana": 4,

"apple": 2,

"orange": 1.5,

"pear": 3

}

stock = {

"banana": 6,

"apple": 0,

"orange": 32,

"pear": 15

}

**Output:**

None

**Keeping Track of the Produce**

Now that you have all of your product info, you should print out all of your inventory information.

once = {'a': 1, 'b': 2}

twice = {'a': 2, 'b': 4}

for key in once:

print "Once: %s" % once[key]

print "Twice: %s" % twice[key]

1. In the above example, we create two dictionaries, once and twice, that have the same keys.
2. Because we know that they have the same keys, we can loop through one dictionary and print values from both once and twice.

Instructions

1. Loop through each key in prices.
2. Like the example above, for each key, print out the key along with its price and stock information. Print the answer in the following format:

apple

price: 2

stock: 0

Like the example above, because you know that the prices and stock dictionary have the same keys, you can access the stock dictionary while you are looping through prices.

When you're printing, you can use the syntax from the example above.

**Script.py**

prices = {

"banana": 4,

"apple": 2,

"orange": 1.5,

"pear": 3

}

stock = {

"banana": 6,

"apple": 0,

"orange": 32,

"pear": 15

}

for key in prices:

print key

print "price: %s" % prices[key]

print "stock: %s" % stock[key]

**Output:**

orange

price: 1.5

stock: 32

pear

price: 3

stock: 15

banana

price: 4

stock: 6

apple

price: 2

stock: 0

**Something of Value**

For paperwork and accounting purposes, let's record the total value of your inventory. It's nice to know what we're worth!

Instructions

Let's determine how much money you would make if you sold all of your food.

1. Create a variable called total and set it to zero.
2. Loop through the prices dictionaries.
3. For each key in prices, multiply the number in prices by the number in stock. Print that value into the console and then add it to total.
4. Finally, outside your loop, print total.

**Script.py**

prices = {

"banana" : 4,

"apple" : 2,

"orange" : 1.5,

"pear" : 3,

}

stock = {

"banana" : 6,

"apple" : 0,

"orange" : 32,

"pear" : 15,

}

total = 0

for key in prices:

print key

print "price: %s" % prices[key]

print "stock: %s" % stock[key]

total = 0

for key in prices:

total = total + prices[key] \* stock[key]

print total

**Output:**

orange

price: 1.5

stock: 32

pear

price: 3

stock: 15

banana

price: 4

stock: 6

apple

price: 2

stock: 0

117.0

**Shopping at the Market**

Great work! Now we're going to take a step back from the management side and take a look through the eyes of the shopper.

In order for customers to order online, we are going to have to make a consumer interface. Don't worry: it's easier than it sounds!

Instructions

First, make a **list** called groceries with the values "banana","orange", and "apple".

**Script.py**

groceries = ["banana", "orange", "apple"]

**Output:**

None

**Making a Purchase**

Good! Now you're going to need to know how much you’re paying for all of the items on your grocery list.

def sum(numbers):

total = 0

for number in numbers:

total += number

return total

n = [1, 2, 5, 10, 13]

print sum(n)

1. In the above example, we first define a function called sum with an argument numbers.
2. We initialize the variable total that we will use as our running sum.
3. For each number in the list, we add that number to the running sum total.
4. At the end of the function, we return the running sum.
5. After the function, we create, n, a list of numbers.
6. Finally, we call the sum(numbers) function with the variable n and print the result.

Instructions

1. Define a function compute\_bill that takes one argument food as input.
2. In the function, create a variable total with an initial value of zero.
3. For each item in the food list, add the price of that item to total.
4. Finally, return the total.

Ignore whether or not the item you're billing for is in stock.

Note that your function should work for **any** food list.

**Script.py**

shopping\_list = ["banana", "orange", "apple"]

stock = {

"banana": 6,

"apple": 0,

"orange": 32,

"pear": 15

}

prices = {

"banana": 4,

"apple": 2,

"orange": 1.5,

"pear": 3

}

# Write your code below!

def compute\_bill(food):

total = 0

for item in food:

total += prices[item]

return total

print compute\_bill(shopping\_list)

**Output:**

7.5

**Stocking Out**

Now you need your compute\_bill function to take the stock/inventory of a particular item into account when computing the cost.

Ultimately, if an item isn't in stock, then it shouldn't be included in the total. You can't buy or sell what you don't have!

Instructions

Make the following changes to your compute\_bill function:

1. While you loop through each item of food, only add the price of the item to total if the item's stock count is greater than zero.
2. If the item is in stock and after you add the price to the total, subtract one from the item's stock count.

**Script.py**

shopping\_list = ["banana", "orange", "apple"]

stock = {

"banana": 6,

"apple": 0,

"orange": 32,

"pear": 15

}

prices = {

"banana": 4,

"apple": 2,

"orange": 1.5,

"pear": 3

}

# Write your code below!

def compute\_bill(food):

total = 0

for item in food:

if stock[item] > 0:

total += prices[item]

stock[item] = stock[item] - 1

return total

**Output:**

None

**Let's Check Out!**

Perfect! You've done a great job with lists and dictionaries in this project. You've practiced:

* Using for loops with lists and dictionaries
* Writing functions with loops, lists, and dictionaries
* Updating data in response to changes in the environment (for instance, decreasing the number of bananas in stock by 1 when you sell one).

Thanks for shopping at the Codecademy supermarket!

Instructions

Click Save & Submit Code to finish this course.

**Script.py**

shopping\_list = ["banana", "orange", "apple"]

stock = {

"banana": 6,

"apple": 0,

"orange": 32,

"pear": 15

}

prices = {

"banana": 4,

"apple": 2,

"orange": 1.5,

"pear": 3

}

# Write your code below!

def compute\_bill(food):

total = 0

for item in food:

if stock[item] > 0:

total += prices[item]

stock[item] = stock[item] - 1

return total

print compute\_bill(shopping\_list)

**Output:**

5.5

**Advanced Topics in Python**

**Iterators for Dictionaries**

Let's start with iterating over a dictionary. Recall that a dictionary is just a collection of keys and values.

d = {

"Name": "Guido",

"Age": 56,

"BDFL": True

}

print d.items()

# => [('BDFL', True), ('Age', 56), ('Name', 'Guido')]

Note that the items() function doesn't return key/value pairs in any specific order. (For more on this, see the **Hint**.)

Instructions

1. Create your own Python dictionary, my\_dict, in the editor to the right with two or three key/value pairs.
2. Then, print the result of calling the my\_dict.items().

**Script.py**

my\_dict = {

"Name": "Mathew",

"Age": 45,

"Job": "Designer"

}

print my\_dict.items()

**Output:**

[('Age', 45), ('Name', 'Mathew'), ('Job', 'Designer')]

**keys() and values()**

Whereas items() returns an array of *tuples* with each tuple consisting of a key/value pair from the dictionary:

* The keys() function returns an array of the dictionary's keys, and
* The values() function returns an array of the dictionary's values.

Again, these functions will not return the keys or values from the dictionary in any specific order.

* You can think of a tuple as an immutable (that is, unchangeable) list (though this is an oversimplification); tuples are surrounded by ()s and can contain any data type.

Instructions

Remove your call to items() and replace it with a call to keys() and a call to values(), each on its own line. Make sure to print both!

**Script.py**

my\_dict = {

"Name": "Mathew",

"Age": 45,

"Job": "Designer"

}

print my\_dict.keys()

print my\_dict.values()

**Output:**

['Age', 'Name', 'Job']

[45, 'Mathew', 'Designer']

**The 'in' Operator**

For iterating over lists, tuples, dictionaries, and strings, Python also includes a special keyword: in. You can use in very intuitively, like so:

for number in range(5):

print number,

d = { "name": "Eric", "age": 26 }

for key in d:

print key, d[key],

for letter in "Eric":

print letter, # note the comma!

1. In the example above, first we create and iterate through a range, printing out 0 1 2 3 4. Note that the trailing comma ensures that we keep printing on the same line.
2. Next, we create a dictionary and iterate through, printing out age 26 name Eric. Dictionaries have no specific order.
3. Finally, we iterate through the letters of a string, printing out E r i c.

Instructions

For each key in my\_dict: print out the key , then a space, then the value stored by that key.

(*You should use print a, b rather than print a + " " + b.*)

**Script.py**

my\_dict = {

"Name": "Mathew",

"Age": 45,

"Job": "Designer"

}

for key in my\_dict:

print key, my\_dict[key]

print my\_dict.items()

**Output:**

Age 45

Name Mathew

Job Designer

[('Age', 45), ('Name', 'Mathew'), ('Job', 'Designer')]

**Building Lists**

Let's say you wanted to build a list of the numbers from 0 to 50 (inclusive). We could do this pretty easily:

my\_list = range(51)

But what if we wanted to generate a list according to some logic—for example, a list of all the even numbers from 0 to 50?

Python's answer to this is the **list comprehension**. List comprehensions are a powerful way to generate lists using the for/in and if keywords we've learned.

Instructions

Check out the list comprehension example in the editor. When you're pretty sure you know what it'll do, click Save & Submit Code to see it in action.

**Script.py**

evens\_to\_50 = [i for i in range(51) if i % 2 == 0]

print evens\_to\_50

**Output:**

[0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44, 46, 48, 50]

**List Comprehension Syntax**

Here's a simple example of list comprehension syntax:

new\_list = [x for x in range(1,6)]

# => [1, 2, 3, 4, 5]

This will create a new\_list populated by the numbers one to five. If you want those numbers doubled, you could use:

doubles = [x\*2 for x in range(1,6)]

# => [2, 4, 6, 8, 10]

And if you only wanted the doubled numbers that are evenly divisible by three:

doubles\_by\_3 = [x\*2 for x in range(1,6) if (x\*2)%3 == 0]

# => [6]

Instructions

1. Use a list comprehension to build a list called even\_squares in the editor.
2. Your even\_squares list should include the squares of the even numbers between 1 to 11. Your list should start [4, 16, 36...] and go from there.

**Script.py**

doubles\_by\_3 = [x\*2 for x in range(1,6) if (x\*2) % 3 == 0]

# Complete the following line. Use the line above for help.

even\_squares = [x\*\*2 for x in range(1,11) if x % 2 == 0]

print even\_squares

**Output:**

[4, 16, 36, 64, 100]

**Now You Try!**

Great work! Now it's time for you to create a list comprehension all on your own.

c = ['C' for x in range(5) if x < 3]

print c

The example above creates and prints out a list containing ['C', 'C', 'C'].

Instructions

1. Use a list comprehension to create a list, cubes\_by\_four.
2. The comprehension should consist of the cubes of the numbers 1 through 10 only if the cube is evenly divisible by four.
3. Finally, print that list to the console.

Note that in this case, the *cubed* number should be evenly divisible by 4, not the original number.

**Script.py**

cubes\_by\_four = [x\*\*3 for x in range(1,11) if (x\*\*3) % 4 == 0]

print cubes\_by\_four

**Output:**

[8, 64, 216, 512, 1000]

**List Slicing Syntax**

Sometimes we only want part of a Python list. Maybe we only want the first few elements; maybe we only want the last few. Maybe we want every other element!

List slicing allows us to access elements of a list in a concise manner. The syntax looks like this:

[start:end:stride]

Where start describes where the slice starts (inclusive), end is where it ends (exclusive), and stride describes the space between items in the sliced list. For example, a stride of 2 would select every other item from the original list to place in the sliced list.

Instructions

We've generated a list with a list comprehension in the editor to the right, and we're about to print a selection from the list using list slicing. Can you guess what will be printed out? Click Save & Submit Code when you think you know!

**Script.py**

l = [i \*\* 2 for i in range(1, 11)]

# Should be [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]

print l[2:9:2]

**Output:**

[9, 25, 49, 81]

**Omitting Indices**

If you don't pass a particular index to the list slice, Python will pick a default.

to\_five = ['A', 'B', 'C', 'D', 'E']

print to\_five[3:]

# prints ['D', 'E']

print to\_five[:2]

# prints ['A', 'B']

print to\_five[::2]

# print ['A', 'C', 'E']

1. The default starting index is 0.
2. The default ending index is the end of the list.
3. The default stride is 1.

Instructions

1. Use list slicing to print out every odd element of my\_list from start to finish.
2. Omit the start and end index. You only need to specify a stride.
3. Check the Hint if you need help.

**Script.py**

my\_list = range(1, 11) # List of numbers 1 - 10

# Add your code below!

print my\_list[::2]

**Output:**

[1, 3, 5, 7, 9]

**Reversing a List**

We have seen that a positive stride progresses through the list from left to right.

A *negative* stride progresses through the list from right to left.

letters = ['A', 'B', 'C', 'D', 'E']

print letters[::-1]

In the example above, we print out ['E', 'D', 'C', 'B', 'A'].

Instructions

1. Create a variable called backwards and set it equal to the reversed version of my\_list.
2. Make sure to reverse the list in the editor by passing your list slice a negative stride, like in the example above.

**Script.py**

my\_list = range(1, 11)

# Add your code below!

backwards = my\_list[::-1]

print backwards

**Output:**

[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]

**Stride Length**

A positive stride length traverses the list from left to right, and a negative one traverses the list from right to left.

Further, a stride length of 1 traverses the list "by ones," a stride length of 2 traverses the list "by twos," and so on.

Instructions

Create a variable, backwards\_by\_tens, and set it equal to the result of going backwards through to\_one\_hundred by tens. Go ahead and print backwards\_by\_tens to the console.

**Script.py**

to\_one\_hundred = range(101)

# Add your code below!

backwards\_by\_tens = to\_one\_hundred[::-10]

print backwards\_by\_tens

**Output:**

[100, 90, 80, 70, 60, 50, 40, 30, 20, 10, 0]

**Practice Makes Perfect**

Great work! See? This list slicing business is pretty straightforward.

Let's do one more, just to prove you really know your stuff.

Instructions

1. Create a list, to\_21, that's just the numbers from 1 to 21, inclusive.
2. Create a second list, odds, that contains only the odd numbers in the to\_21 list (1, 3, 5, and so on). Use list slicing for this one instead of a list comprehension.
3. Finally, create a third list, middle\_third, that's equal to the middle third of to\_21, from 8 to 14, inclusive.

**Script.py**

to\_21 = range(1,22)

odds = to\_21[::2]

middle\_third = to\_21[7:14:1]

print to\_21

print odds

print middle\_third

**Output:**

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21]

[1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21]

[8, 9, 10, 11, 12, 13, 14]

**Anonymous Functions**

One of the more powerful aspects of Python is that it allows for a style of programming called **functional programming**, which means that you're allowed to pass functions around just as if they were variables or values. Sometimes we take this for granted, but not all languages allow this!

Check out the code at the right. See the lambda bit? Typing

lambda x: x % 3 == 0

Is the same as

def by\_three(x):

return x % 3 == 0

Only we don't need to actually give the function a name; it does its work and returns a value without one. That's why the function the lambda creates is an **anonymous function**.

When we pass the lambda to filter, filter uses the lambda to determine what to filter, and the second argument (my\_list, which is just the numbers 0 – 15) is the list it does the filtering on.

Instructions

Can you guess what the this code will print to the console? Click Save & Submit Code to see.

**Script.py**

my\_list = range(16)

print filter(lambda x: x % 3 == 0, my\_list)

**Output:**

[0, 3, 6, 9, 12, 15]

**Lambda Syntax**

Lambda functions are defined using the following syntax:

my\_list = range(16)

filter(lambda x: x % 3 == 0, my\_list)

Lambdas are useful when you need a quick function to do some work for you.

If you plan on creating a function you'll use over and over, you're better off using def and giving that function a name.

Instructions

1. Fill in the first part of the filter function with a lambda. The lambda should ensure that only "Python" is returned by the filter.
2. Fill in the second part of the filter function with languages, the list to filter.

**Script.py**

languages = ["HTML", "JavaScript", "Python", "Ruby"]

print filter(lambda x: x == "Python", languages)

**Output:**

['Python']

**Try It!**

All right! Time to test out filter() and lambda expressions.

cubes = [x\*\*3 for x in range(1, 11)]

filter(lambda x: x % 3 == 0, cubes)

The example above is just a reminder of the syntax.

Instructions

1. Create a list, squares, that consists of the squares of the numbers 1 to 10. A list comprehension could be useful here!
2. Use filter() and a lambda expression to print out only the squares that are between 30 and 70 (inclusive).

Script.py

squares = [x\*\*2 for x in range(1,11)]

print filter(lambda x: x >= 30 and x <= 70, squares)

Output:

[36, 49, 64]

**Iterating Over Dictionaries**

First, let's review iterating over a dict.

Instructions

Call the appropriate method on movies such that it will print out all the *items* (hint, hint) in the dictionary—that is, each key and each value.

Script.py

movies = {

"Monty Python and the Holy Grail": "Great",

"Monty Python's Life of Brian": "Good",

"Monty Python's Meaning of Life": "Okay"

}

print movies.items()

Output:

[("Monty Python's Life of Brian", 'Good'), ("Monty Python's Meaning of Life", 'Okay'), ('Monty Python and the Holy Grail', 'Great')]

**Comprehending Comprehensions**

Good! Now let's take another look at list comprehensions.

squares = [x\*\*2 for x in range(5)]

Instructions

Use a list comprehension to create a list, threes\_and\_fives, that consists only of the numbers between 1 and 15 (inclusive) that are evenly divisible by 3 or 5.

**Script.py**

threes\_and\_fives = [x for x in range(1,16) if x % 3 == 0 or x % 5 == 0]

print threes\_and\_fives

**Output:**

[3, 5, 6, 9, 10, 12, 15]

**List Slicing**

Great! Next up: list slicing.

str = "ABCDEFGHIJ"

start, end, stride = 1, 6, 2

str[start:end:stride]

You can think of a Python string as a list of characters.

Instructions

The string in the editor is garbled in two ways:

1. First, our message is backwards;
2. Second, the letter we want is every other letter.

Use list slicing to extract the message and save it to a variable called message.

**Script.py**

garbled = "!XeXgXaXsXsXeXmX XtXeXrXcXeXsX XeXhXtX XmXaX XI"

message = garbled[::-2]

print message

**Output:**

I am the secret message!

**Lambda Expressions**

Last but not least, let's look over some lambdas.

my\_list = range(16)

filter(lambda x: x % 3 == 0, my\_list)

We've given you another (slightly different) garbled. Sort it out with a filter() and a lambda.

Instructions

1. Create a new variable called message.
2. Set it to the result of calling filter() with the appropriate lambda that will filter out the "X"s. The second argument will be garbled.
3. Finally, print your message to the console.

**Script.py**

garbled = "IXXX aXXmX aXXXnXoXXXXXtXhXeXXXXrX sXXXXeXcXXXrXeXt mXXeXsXXXsXaXXXXXXgXeX!XX"

message = filter(lambda x: x is not 'X', garbled)

print message

**Output:**

I am another secret message!

**Introduction to Bitwise Operators**

**Just a Little BIT**

Welcome to an intro level explanation of bitwise operations in Python!

Bitwise operations might seem a little esoteric and tricky at first, but you'll get the hang of them pretty quickly.

**Bitwise operations** are operations that directly manipulate **bits**. In all computers, numbers are represented with bits, a series of zeros and ones. In fact, pretty much everything in a computer is represented by bits. This course will introduce you to the basic bitwise operations and then show you what you can do with them.

Bitwise operators often tend to puzzle and mystify new programmers, so don't worry if you are a little bit confused at first. To be honest, you aren't really going to see bitwise operators in your everyday program. However, they do pop up from time to time, and when they do, you should have a general idea of what is going on.

Instructions

In the editor are the 6 basic bitwise operations. Click Save & Submit Code and see what the console prints out. All of them will be explained in due time!

**Script.py**

print 5 >> 4 # Right Shift

print 5 << 1 # Left Shift

print 8 & 5 # Bitwise AND

print 9 | 4 # Bitwise OR

print 12 ^ 42 # Bitwise XOR

print ~88 # Bitwise NOT

**Output:**

0

10

0

13

38

-89

**Lesson I0: The Base 2 Number System**

When we count, we usually do it in base 10. That means that each place in a number can hold one of ten values, 0-9. In binary we count in base two, where each place can hold one of two values: 0 or 1. The counting pattern is the same as in base 10 except when you carry over to a new column, you have to carry over every time a place goes higher than one (as opposed to higher than 9 in base 10).

For example, the numbers one and zero are the same in base 10 and base 2. But in base 2, once you get to the number 2 you have to carry over the one, resulting in the representation "10". Adding one again results in "11" (3) and adding one again results in "100" (4).

Contrary to counting in base 10, where each decimal place represents a power of 10, each place in a binary number represents a power of two (or a **bit**). The rightmost bit is the 1's bit (two to the zero power), the next bit is the 2's bit (two to the first), then 4, 8, 16, 32, and so on.

The binary number '1010' is 10 in base 2 because the 8's bit and the 2's bit are "on":

8's bit 4's bit 2's bit 1's bit

1 0 1 0

8 + 0 + 2 + 0 = 10

In Python, you can write numbers in binary format by starting the number with 0b. When doing so, the numbers can be operated on like any other number!

Instructions

Take a look at the examples in the editor. Really try to understand this pattern before moving on. Click Save & Submit Code when you're ready to continue.

**Script.py**

print 0b1, #1

print 0b10, #2

print 0b11, #3

print 0b100, #4

print 0b101, #5

print 0b110, #6

print 0b111 #7

print "\*\*\*\*\*\*"

print 0b1 + 0b11

print 0b11 \* 0b11

**Output:**

1 2 3 4 5 6 7

\*\*\*\*\*\*

4

9

**I Can Count to 1100!**

All right! Time to practice counting in binary.

To make sure you've got the hang of it, fill out the rest of the numbers all the way up to twelve. Please **do not** use the str() method or any other outside functions.

Here are a few numbers that will be good to know going forward -

2\*\*0 = 1

2\*\*1 = 2

2\*\*2 = 4

2\*\*3 = 8

2\*\*4 = 16

2\*\*5 = 32

2\*\*6 = 64

2\*\*7 = 128

2\*\*8 = 256

2\*\*9 = 512

2\*\*10 = 1024

You may recognize these numbers. Do you have a 32 or 64 bit system? Does your computer have a 256GB hard drive? Computers think in binary!

Instructions

Fill out the rest of the numbers with their corresponding binary values up to twelve in the editor to the right, using the 0bxxx format.

**Script.py**

one = 0b1

two = 0b10

three = 0b11

four = 0b100

five = 0b101

six = 0b110

seven = 0b111

eight = 0b1000

nine = 0b1001

ten = 0b1010

eleven = 0b1011

twelve = 0b1100

print one, two, three, four, five, six, seven, eight, nine, ten, eleven, twelve

**Output:**

1 2 3 4 5 6 7 8 9 10 11 12

**The bin() Function**

Excellent! The biggest hurdle you have to jump over in order to understand bitwise operators is learning how to count in base 2. Hopefully the lesson should be easier for you from here on out.

There are Python functions that can aid you with bitwise operations. In order to print a number in its binary representation, you can use the bin() function. bin() takes an integer as input and returns the binary representation of that integer in a string. (Keep in mind that after using the bin function, you can no longer operate on the value like a number.)

You can also represent numbers in base 8 and base 16 using the oct() and hex() functions. (We won't be dealing with those here, however.)

Instructions

We've provided an example of the bin function in the editor. Go ahead and use print and bin() to print out the binary representations of the numbers 2 through 5, each on its own line.

**Script.py**

print bin(1)

print bin(2)

print bin(3)

print bin(4)

print bin(5)

**Output:**

0b1

0b10

0b11

0b100

0b101

**int()'s Second Parameter**

Python has an int() function that you've seen a bit of already. It can turn non-integer input into an integer, like this:

int("42")

# ==> 42

What you might not know is that the int function actually has an optional second parameter.

int("110", 2)

# ==> 6

When given a string containing a number and the base that number is in, the function will return the value of that number converted to base ten.

Instructions

In the console are several different ways that you can use the int function's second parameter.

On [line 7](javascript:void(0)), use int to print the base 10 equivalent of the binary number 11001001.

**Script.py**

print int("1",2)

print int("10",2)

print int("111",2)

print int("0b100",2)

print int(bin(5),2)

# Print out the decimal equivalent of the binary 11001001.

print int('11001001',2)

**Output:**

1

2

7

4

5

201

**Slide to the Left! Slide to the Right!**

The next two operations we are going to talk about are the left and right shift bitwise operators. These operators work by shifting the bits of a number over by a designated number of slots.

The block below shows how these operators work on the bit level. Note that in the diagram, the shift is always a positive integer:

# Left Bit Shift (<<)

0b000001 << 2 == 0b000100 (1 << 2 = 4)

0b000101 << 3 == 0b101000 (5 << 3 = 40)

# Right Bit Shift (>>)

0b0010100 >> 3 == 0b000010 (20 >> 3 = 2)

0b0000010 >> 2 == 0b000000 (2 >> 2 = 0)

This operation is mathematically equivalent to floor dividing and multiplying by 2 (respectively) for every time you shift, but it's often easier just to think of it as shifting all the 1s and 0s left or right by the specified number of slots.

Note that you can only do bitwise operations on an **integer**. Trying to do them on strings or floats will result in nonsensical output!

Instructions

Shift the variable shift\_right to the right twice (>> 2) and shift the variable shift\_left to the left twice (<< 2). Try to guess what the printed output will be!

**Script.py**

shift\_right = 0b1100 >> 2

shift\_left = 0b1 << 2

# Your code here!

0b1100 >> 2 == 0b11

0b1 >> 2 == 0b100

print bin(shift\_right)

print bin(shift\_left)

**Output:**

0b11

0b100

**A BIT of This AND That**

The bitwise AND (&) operator compares two numbers on a bit level and returns a number where the bits of that number are turned on if the corresponding bits of **both** numbers are 1. For example:

a: 00101010 42

b: 00001111 15

===================

a & b: 00001010 10

As you can see, the 2's bit and the 8's bit are the only bits that are on in both a and b, so a & b only contains those bits. Note that using the & operator can only result in a number that is less than or equal to the smaller of the two values.

So remember, for every given bit in a and b:

0 & 0 = 0

0 & 1 = 0

1 & 0 = 0

1 & 1 = 1

Therefore,

0b111 (7) & 0b1010 (10) = 0b10

which equals two.

Instructions

print out the result of calling bin() on 0b1110 & 0b101.

See if you can guess what the output will be!

**Script.py**

print bin(0b1110 & 0b101)

**Output:**

0b100

**A BIT of This OR That**

The bitwise OR (|) operator compares two numbers on a bit level and returns a number where the bits of that number are turned on if **either** of the corresponding bits of either number are 1. For example:

a: 00101010 42

b: 00001111 15

================

a | b: 00101111 47

Note that the bitwise | operator can only create results that are greater than or equal to the larger of the two integer inputs.

So remember, for every given bit in a and b:

0 | 0 = 0

0 | 1 = 1

1 | 0 = 1

1 | 1 = 1

Meaning

110 (6) | 1010 (10) = 1110 (14)

Instructions

For practice, print out the result of using | on 0b1110 and 0b101 as a binary string. Try to do it on your own without using the | operator if you can help it.

**Script.py**

print bin(0b1110 | 0b101)

**Output:**

0b1111

**This XOR That?**

The XOR (^) or *exclusive or* operator compares two numbers on a bit level and returns a number where the bits of that number are turned on if **either** of the corresponding bits of the two numbers are 1, **but not both**.

a: 00101010 42

b: 00001111 15

================

a ^ b: 00100101 37

Keep in mind that if a bit is off in both numbers, it stays off in the result. Note that XOR-ing a number with itself will always result in 0.

So remember, for every given bit in a and b:

0 ^ 0 = 0

0 ^ 1 = 1

1 ^ 0 = 1

1 ^ 1 = 0

Therefore:

111 (7) ^ 1010 (10) = 1101 (13)

Instructions

For practice, print the result of using ^ on 0b1110 and 0b101 as a binary string. Try to do it on your own without using the ^ operator.

**Script.py**

print bin(0b1110 ^ 0b101)

**Output:**

0b1011

**See? This is NOT That Hard!**

The bitwise NOT operator (~) just flips all of the bits in a single number. What this actually means to the computer is actually very complicated, so we're not going to get into it. Just know that mathematically, this is equivalent to adding one to the number and then making it negative.

And with that, you've seen all of the basic bitwise operators! We'll see what we can do with these in the next section.

Instructions

Click Save & Submit Code and observe what the console prints out.

**Script.py**

print ~1

print ~2

print ~3

print ~42

print ~123

**Output:**

-2

-3

-4

-43

-124

**The Man Behind the Bit Mask**

A **bit mask** is just a variable that aids you with bitwise operations. A bit mask can help you turn specific bits on, turn others off, or just collect data from an integer about which bits are on or off.

num = 0b1100

mask = 0b0100

desired = num & mask

if desired > 0:

print "Bit was on"

In the example above, we want to see if the third bit from the right is on.

1. First, we first create a variable num containing the number 12, or 0b1100.
2. Next, we create a mask with the third bit on.
3. Then, we use a bitwise-and operation to see if the third bit from the right of num is on.
4. If desired is greater than zero, then the third bit of num must have been one.

Instructions

1. Define a function, check\_bit4, with one argument, input, an integer.
2. It should check to see if the fourth bit from the right is on.
3. If the bit is on, return "on" (not print!)
4. If the bit is off, return "off".

Check the Hint for some examples!

Hint

Here are some examples:

check\_bit4(0b1) # ==> "off"

check\_bit4(0b11011) # ==> "on"

check\_bit4(0b1010) # ==> "on"

You'll need to use a mask where all bits are off except for the fourth bit from the right.

**Script.py**

def check\_bit4(input):

mask = 0b1000

if input & mask > 0:

return "on"

else:

return "off"

print check\_bit4(0b1010)

**Output:**

on

**Turn It On**

You can also use masks to turn a bit in a number on using |. For example, let's say I want to make sure the rightmost bit of number a is turned on. I could do this:

a = 0b110 # 6

mask = 0b1 # 1

desired = a | mask # 0b111, or 7

Using the bitwise | operator will turn a corresponding bit on if it is off and leave it on if it is already on.

Instructions

In the editor is a variable, a. Use a bitmask and the value a in order to achieve a result where the third bit from the right of a is turned on. Be sure to print your answer as a bin() string!

**Script.py**

a = 0b10111011

mask = 0b100

desired = a | mask

print bin(desired)

**Output:**

0b10111111

**Just Flip Out**

Using the XOR (^) operator is very useful for flipping bits. Using ^ on a bit with the number one will return a result where that bit is flipped.

For example, let's say I want to flip all of the bits in a. I might do this:

a = 0b110 # 6

mask = 0b111 # 7

desired = a ^ mask # 0b1

Instructions

In the editor is the 8 bit variable a. Use a bitmask and the value a in order to achieve a result where all of the bits in a are flipped. Be sure to print your answer as a bin() string!

**Script.py**

a = 0b11101110

mask = 0b11111111

desired = a ^ mask

print bin(desired)

**Output:**

0b10001

**Slip and Slide**

Finally, you can also use the left shift (<<) and right shift (>>) operators to slide masks into place.

a = 0b101

# Tenth bit mask

mask = (0b1 << 9) # One less than ten

desired = a ^ mask

Let's say that I want to turn on the 10th bit from the right of the integer a.

Instead of writing out the entire number, we slide a bit over using the << operator.

We use 9 because we only need to slide the mask nine places over from the first bit to reach the tenth bit.

Instructions

1. Define a function called flip\_bit that takes the inputs (number, n).
2. Flip the nth bit (with the ones bit being the first bit) and store it in result.
3. Return the result of calling bin(result).

**Script.py**

def flip\_bit(number,n):

mask = (0b1<<n-1)

desired = number^mask

return bin(desired)

print flip\_bit(0b10100101, 4)

**Output:**

0b10101101

**Introduction to Classes**

**Why Use Classes?**

Python is an object-oriented programming language, which means it manipulates programming constructs called **objects**. You can think of an object as a single data structure that contains data as well as functions; functions of objects are called **methods**. For example, any time you call

len("Eric")

Python is checking to see whether the string object you passed it has a length, and if it does, it returns the value associated with that **attribute**. When you call

my\_dict.items()

Python checks to see if my\_dict has an items() method (which all dictionaries have) and executes that method if it finds it.

But what makes "Eric" a string and my\_dict a dictionary? The fact that they're instances of the str and dict classes, respectively. A class is just a way of organizing and producing objects with similar attributes and methods.

Instructions

Check out the code in the editor to the right. We've defined our *own* class, Fruit, and created a lemon instance.

When you're ready, click Save & Submit Code to get started creating classes and objects of your own.

**Script.py**

class Fruit(object):

"""A class that makes various tasty fruits."""

def \_\_init\_\_(self, name, color, flavor, poisonous):

self.name = name

self.color = color

self.flavor = flavor

self.poisonous = poisonous

def description(self):

print "I'm a %s %s and I taste %s." % (self.color, self.name, self.flavor)

def is\_edible(self):

if not self.poisonous:

print "Yep! I'm edible."

else:

print "Don't eat me! I am super poisonous."

lemon = Fruit("lemon", "yellow", "sour", False)

lemon.description()

lemon.is\_edible()

**Output:**

I'm a yellow lemon and I taste sour.

Yep! I'm edible.

**Class Syntax**

A basic class consists only of the class keyword, the name of the class, and the class from which the new class **inherits** in parentheses. (We'll get to inheritance soon.) For now, our classes will inherit from the object class, like so:

class NewClass(object):

# Class magic here

This gives them the powers and abilities of a Python object. By convention, user-defined Python class names start with a capital letter.

Instructions

Create a class called Animal in the editor. For now, in the body of your class, use the pass keyword. (pass doesn't do anything, but it's useful as a placeholder in areas of your code where Python expects an expression.)

**Script.py**

class Animal(object):

pass

**Output:**

None

**Classier Classes**

We'd like our classes to do more than... well, *nothing*, so we'll have to replace our pass with something else.

You may have noticed in our example back in the first exercise that we started our class definition off with an odd-looking function: \_\_init\_\_(). This function is required for classes, and it's used to **initialize** the objects it creates. \_\_init\_\_() always takes at least one argument, self, that refers to the object being created. You can think of \_\_init\_\_() as the function that "boots up" each object the class creates.

Instructions

Remove the pass statement in your class definition, then go ahead and define an \_\_init\_\_() function for your Animal class. Pass it the argument self for now; we'll explain how this works in greater detail in the next section. Finally, put the pass into the body of the \_\_init\_\_() definition, since it will expect an indented block.

**Script.py**

class Animal(object):

def\_\_init\_\_(self):

pass

**Output:**

None

**Let's Not Get Too Selfish**

Excellent! Let's make one more tweak to our class definition, then go ahead and **instantiate** (create) our first object.

So far, \_\_init\_\_() only takes one parameter: self. This is a Python convention; there's nothing magic about the word self. However, it's overwhelmingly common to use self as the first parameter in \_\_init\_\_(), so you should do this so that other people will understand your code.

The part that *is* magic is the fact that self is the *first* parameter passed to \_\_init\_\_(). Python will use the first parameter that \_\_init\_\_() receives to refer to the object being created; this is why it's often called self, since this parameter gives the object being created its identity.

Instructions

Let's do two things in the editor:

1. Pass \_\_init\_\_() a second parameter, name.
2. In the body of \_\_init\_\_(), let the function know that name refers to the created object's name by typing self.name = name. (This will become crystal clear in the next section.)

**Script.py**

class Animal(object):

def \_\_init\_\_(self, name):

self.name = name

**Output:**

None

**Instantiating Your First Object**

Perfect! Now we're ready to start creating objects.

We can access attributes of our objects using **dot notation** Here's how it works:

class Square(object):

def \_\_init\_\_(self):

self.sides = 4

my\_shape = Square()

print my\_shape.sides

1. First we create a class named Square with an attribute sides.
2. Outside the class definition, we create a new instance of Square named my\_shape and access that attribute using my\_shape.sides.

Instructions

1. Outside the Animal class definition, create a variable named zebra and set it equal to Animal("Jeffrey").
2. Then print out zebra's name.

**Script.py**

class Animal(object):

def \_\_init\_\_(self, name):

self.name = name

zebra = Animal("Jeffrey")

print zebra.name

**Output:**

Jeffrey

**More on \_\_init\_\_() and self**

Now that you're starting to understand how classes and objects work, it's worth delving a bit more into \_\_init\_\_() and self. They can be confusing!

As mentioned, you can think of \_\_init\_\_() as the method that "boots up" a class' instance object: the init bit is short for "initialize."

The first argument \_\_init\_\_() gets is used to refer to the instance object, and by convention, that argument is called self. If you add additional arguments—for instance, a name and age for your animal—setting each of those equal to self.name and self.age in the body of \_\_init\_\_() will make it so that when you create an instance object of your Animal class, you need to give each instance a name and an age, and those will be associated with the particular instance you create.

Instructions

Check out the examples in the editor. See how \_\_init\_\_() "boots up" each object to expect a name and an age, then uses self.name and self.age to assign those names and ages to each object? Add a third attribute, is\_hungry to \_\_init\_\_(), and click Save & Submit Code to see the results.

**Script.py**

# Class definition

class Animal(object):

"""Makes cute animals."""

# For initializing our instance objects

def \_\_init\_\_(self, name, age, is\_hungry):

self.name = name

self.age = age

self.is\_hungry = is\_hungry

# Note that self is only used in the \_\_init\_\_()

# function definition; we don't need to pass it

# to our instance objects.

zebra = Animal("Jeffrey", 2, True)

giraffe = Animal("Bruce", 1, False)

panda = Animal("Chad", 7, True)

print zebra.name, zebra.age, zebra.is\_hungry

print giraffe.name, giraffe.age, giraffe.is\_hungry

print panda.name, panda.age, panda.is\_hungry

**Output:**

Jeffrey 2 True

Bruce 1 False

Chad 7 True

**Class Scope**

Another important aspect of Python classes is **scope**. The scope of a variable is the context in which it's visible to the program.

It may surprise you to learn that not all variables are accessible to all parts of a Python program at all times. When dealing with classes, you can have variables that are available everywhere (**global variables**), variables that are only available to members of a certain class (**member variables**), and variables that are only available to particular instances of a class (**instance variables**).

The same goes for functions: some are available everywhere, some are only available to members of a certain class, and still others are only available to particular instance objects.

Instructions

Check out the code in the editor. Note that each individual animal gets its own name and age (since they're all initialized individually), but they all have access to the member variable is\_alive, since they're all members of the Animal class. Click Save & Submit Code to see the output!

**Script.py**

class Animal(object):

"""Makes cute animals."""

is\_alive = True

def \_\_init\_\_(self, name, age):

self.name = name

self.age = age

zebra = Animal("Jeffrey", 2)

giraffe = Animal("Bruce", 1)

panda = Animal("Chad", 7)

print zebra.name, zebra.age, zebra.is\_alive

print giraffe.name, giraffe.age, giraffe.is\_alive

print panda.name, panda.age, panda.is\_alive

**Output:**

Jeffrey 2 True

Bruce 1 True

Chad 7 True

**A Methodical Approach**

When a class has its own functions, those functions are called **methods**. You've already seen one such method: \_\_init\_\_(). But you can also define your own methods!

Instructions

Add a method, description, to your Animal class. Using two separate print statements, it should print out the name and age of the animal it's called on. Then, create an instance of Animal, hippo (with whatever name and age you like), and call its description method.

**Script.py**

**class Animal(object):**

**"""Makes cute animals."""**

**is\_alive = True**

**def \_\_init\_\_(self, name, age):**

**self.name = name**

**self.age = age**

**# Add your method here!**

**def description(self):**

**print self.name**

**print self.age**

**print hippo.description()**

**hippo = Animal("Gus", 2)**

**Output:**

None

**They're Multiplying!**

A class can have any number of **member variables**. These are variables that are available to all members of a class.

hippo = Animal("Jake", 12)

cat = Animal("Boots", 3)

print hippo.is\_alive

hippo.is\_alive = False

print hippo.is\_alive

print cat.is\_alive

1. In the example above, we create two instances of an Animal.
2. Then we print out True, the default value stored in hippo's is\_alive member variable.
3. Next, we set that to False and print it out to make sure.
4. Finally, we print out True, the value stored in cat's is\_alive member variable. We only changed the variable in hippo, not in cat.

Let's add another member variable to Animal.

Instructions

1. After [line 3](javascript:void(0)), add a second member variable called health that contains the string "good".
2. Then, create two new Animals: sloth and ocelot. (Give them whatever names and ages you like.)
3. Finally, on three separate lines, print out the health of your hippo, sloth, and ocelot.

**Script.py**

class Animal(object):

"""Makes cute animals."""

is\_alive = True

health = "good"

def \_\_init\_\_(self, name, age):

self.name = name

self.age = age

# Add your method here!

def description(self):

print self.name

print self.age

hippo = Animal("Gus", 2)

sloth = Animal("Pete", 4)

ocelot = Animal("Ramona", 3)

print hippo.description()

print hippo.health

print sloth.health

print ocelot.health

**Output:**

Gus

2

None

good

good

good

**It's Not All Animals and Fruits**

Classes like Animal and Fruit make it easy to understand the concepts of classes and instances, but you probably won't see many zebras or lemons in real-world programs.

However, classes and objects are often used to model real-world objects. The code in the editor is a more realistic demonstration of the kind of classes and objects you might find in commercial software. Here we have a basic ShoppingCart class for creating shopping cart objects for website customers; though basic, it's similar to what you'd see in a real program.

Instructions

Create an instance of ShoppingCart called my\_cart. Initialize it with any values you like, then use the add\_item method to add an item to your cart.

**Script.py**

class ShoppingCart(object):

"""Creates shopping cart objects

for users of our fine website."""

items\_in\_cart = {}

def \_\_init\_\_(self, customer\_name):

self.customer\_name = customer\_name

def add\_item(self, product, price):

"""Add product to the cart."""

if not product in self.items\_in\_cart:

self.items\_in\_cart[product] = price

print product + " added."

else:

print product + " is already in the cart."

def remove\_item(self, product):

"""Remove product from the cart."""

if product in self.items\_in\_cart:

del self.items\_in\_cart[product]

print product + " removed."

else:

print product + " is not in the cart."

my\_cart = ShoppingCart("Ralph")

my\_cart.add\_item("book", 4)

**Output:**

book added.

**Warning: Here Be Dragons**

**Inheritance** is a tricky concept, so let's go through it step by step.

Inheritance is the process by which one class takes on the attributes and methods of another, and it's used to express an **is-a** relationship. For example, a Panda **is a** bear, so a Panda class could inherit from a Bear class. However, a Toyota is not a Tractor, so it shouldn't inherit from the Tractor class (even if they have a lot of attributes and methods in common). Instead, both Toyota and Tractor could ultimately inherit from the same Vehicle class.

Instructions

Check out the code in the editor. We've defined a class, Customer, as well as a ReturningCustomer class that inherits from Customer. Note that we don't define the display\_cart method in the body of ReturningCustomer, but it will still have access to that method via inheritance. Click Save & Submit Code to see for yourself!

**Script.py**

class Customer(object):

"""Produces objects that represent customers."""

def \_\_init\_\_(self, customer\_id):

self.customer\_id = customer\_id

def display\_cart(self):

print "I'm a string that stands in for the contents of your shopping cart!"

class ReturningCustomer(Customer):

"""For customers of the repeat variety."""

def display\_order\_history(self):

print "I'm a string that stands in for your order history!"

monty\_python = ReturningCustomer("ID: 12345")

monty\_python.display\_cart()

monty\_python.display\_order\_history()

**Output:**

I'm a string that stands in for the contents of your shopping cart!

I'm a string that stands in for your order history!

**Inheritance Syntax**

In Python, inheritance works like this:

class DerivedClass(BaseClass):

# code goes here

where DerivedClass is the new class you're making and BaseClass is the class from which that new class inherits.

Instructions

On [lines 1-4](javascript:void(0)), we've created a class named Shape.

1. Create your own class, Triangle, that inherits from Shape, like this:
2. class Triangle(Shape):
3. # code goes here
4. Inside the Triangle class, write an \_\_init\_\_() function that takes four arguments: self, side1, side2, and side3.
5. Inside the \_\_init\_\_() function, set self.side1 = side1, self.side2 = side2, and self.side3 = side3.

Your code should look something like this:

class Triangle(Shape):

def \_\_init\_\_(self, side1, side2, side3):

self.side1 = side1

self.side2 = side2

self.side3 = side3

**Script.py**

class Shape(object):

"""Makes shapes!"""

def \_\_init\_\_(self, number\_of\_sides):

self.number\_of\_sides = number\_of\_sides

# Add your Triangle class below!

class Triangle(Shape):

def \_\_init\_\_(self, side1, side2, side3):

self.side1 = side1

self.side2 = side2

self.side3 = side3

**Output:**

None

**Override!**

Sometimes you'll want one class that inherits from another to not only take on the methods and attributes of its parent, but to **override** one or more of them.

class Employee(object):

def \_\_init\_\_(self, name):

self.name = name

def greet(self, other):

print "Hello, %s" % other.name

class CEO(Employee):

def greet(self, other):

print "Get back to work, %s!" % other.name

ceo = CEO("Emily")

emp = Employee("Steve")

emp.greet(ceo)

# Hello, Emily

ceo.greet(emp)

# Get back to work, Steve!

Rather than have a separate greet\_underling method for our CEO, we **override** (or re-create) the greet method on top of the base Employee.greet method. This way, we don't need to know what type of Employee we have before we greet another Employee.

Instructions

1. Create a new class, PartTimeEmployee, that inherits from Employee.
2. Give your derived class a calculate\_wage method that overrides Employee's. It should take self and hours as arguments.
3. Because PartTimeEmployee.calculate\_wage overrides Employee.calculate\_wage, it still needs to set self.hours = hours.
4. It should return the part-time employee's number of hours worked multiplied by 12.00 (that is, they get $12.00 per hour instead of $20.00).

**Script.py**

class Employee(object):

"""Models real-life employees!"""

def \_\_init\_\_(self, employee\_name):

self.employee\_name = employee\_name

def calculate\_wage(self, hours):

self.hours = hours

return hours \* 20.00

# Add your code below!

class PartTimeEmployee(Employee):

def calculate\_wage(self, hours):

self.hours = hours

return hours \* 12.00

**Output:**

None

**This Looks Like a Job For...**

On the flip side, sometimes you'll be working with a derived class (or **subclass**) and realize that you've overwritten a method or attribute defined in that class' base class (also called a **parent** or **superclass**) that you actually need. Have no fear! You can directly access the attributes or methods of a superclass with Python's built-in super call.

The syntax looks like this:

class Derived(Base):

def m(self):

return super(Derived, self).m()

Where m() is a method from the base class.

Instructions

First, inside your PartTimeEmployee class:

1. Add a new method called full\_time\_wage with the arguments self and hours.
2. That method should return the result of a super call to the calculate\_wage method of PartTimeEmployee's parent class. Use the example above for help.

Then, after your class:

1. Create an instance of the PartTimeEmployee class called milton. Don't forget to give it a name.
2. Finally, print out the result of calling his full\_time\_wage method. You should see his wage printed out at $20.00 per hour! (That is, for 10 hours, the result should be 200.00.)

**Script.py**

class Employee(object):

"""Models real-life employees!"""

def \_\_init\_\_(self, employee\_name):

self.employee\_name = employee\_name

def calculate\_wage(self, hours):

self.hours = hours

return hours \* 20.00

# Add your code below!

class PartTimeEmployee(Employee):

def calculate\_wage(self, hours):

self.hours = hours

return hours \* 12.00

def full\_time\_wage(self, hours):

return super(PartTimeEmployee, self).calculate\_wage(hours)

milton = PartTimeEmployee("Milton")

print milton.full\_time\_wage

**Output:**

<bound method PartTimeEmployee.full\_time\_wage of <PartTimeEmployee object at 0x7f7cbb445bd0>>

**Class Basics**

First things first: let's create a class to work with.

Instructions

Create a class, Triangle. Its \_\_init\_\_() method should take self, angle1, angle2, and angle3 as arguments. Make sure to set these appropriately in the body of the \_\_init\_\_() method (see the Hint for more).

**Script.py**

class Triangle(object):

def \_\_init\_\_(self, angle1, angle2, angle3):

self.angle1 = angle1

self.angle2 = angle2

self.angle3 = angle3

**Output:**

None

**Class It Up**

Great! Now let's add a member variable and a method to our class.

Instructions

Inside the Triangle class:

1. Create a variable named number\_of\_sides and set it equal to 3.
2. Create a method named check\_angles. The sum of a triangle's three angles should return True if the sum of self.angle1, self.angle2, and self.angle3 is equal 180, and False otherwise.

**Script.py**

class Triangle(object):

number\_of\_sides = 3

def \_\_init\_\_(self, angle1, angle2, angle3):

self.angle1 = angle1

self.angle2 = angle2

self.angle3 = angle3

def check\_angles(self):

if self.angle1 + self.angle2 + self.angle3 == 180:

return True

else:

return False

**Output:**

None

**Instantiate an Object**

Let's go ahead and create an instance of our Triangle class.

Instructions

1. Create a variable named my\_triangle and set it equal to a new instance of your Triangle class. Pass it three angles that sum to 180 (e.g. 90, 30, 60).
2. Print out my\_triangle.number\_of\_sides
3. Print out my\_triangle.check\_angles()

**Script.py**

class Triangle(object):

number\_of\_sides = 3

def \_\_init\_\_(self, angle1, angle2, angle3):

self.angle1 = angle1

self.angle2 = angle2

self.angle3 = angle3

def check\_angles(self):

if self.angle1 + self.angle2 + self.angle3 == 180:

return True

else:

return False

my\_triangle = Triangle(90, 30, 60)

print my\_triangle.number\_of\_sides

print my\_triangle.check\_angles()

**Output:**

3

True

**Inheritance**

Finally, let's create an Equilateral class that inherits from our Triangle class. (An equilateral triangle is a triangle whose angles are all 60˚, which also means that its three sides are equal in length.)

Instructions

1. Create a class named Equilateral that inherits from Triangle.
2. Inside Equilateral, create a member variable named angle and set it equal to 60.
3. Create an \_\_init\_\_() function with only the parameter self, and set self.angle1, self.angle2, and self.angle3 equal to self.angle (since an equilateral triangle's angles will always be 60˚).

**Script.py**

class Triangle(object):

number\_of\_sides = 3

def \_\_init\_\_(self, angle1, angle2, angle3):

self.angle1 = angle1

self.angle2 = angle2

self.angle3 = angle3

def check\_angles(self):

if self.angle1 + self.angle2 + self.angle3 == 180:

return True

else:

return False

my\_triangle = Triangle(90, 30, 60)

print my\_triangle.number\_of\_sides

print my\_triangle.check\_angles()

class Equilateral(Triangle):

angle = 60

def \_\_init\_\_(self):

self.angle1 = self.angle

self.angle2 = self.angle

self.angle3 = self.angle

**Output:**

3

True

**Classes**

**Class basics**

Classes can be very useful for storing complicated objects with their own methods and variables. Defining a class is much like defining a function, but we use the class keyword instead. We also use the word object in parentheses because we want our classes to **inherit** the object class. This means that our class has all the properties of an object, which is the simplest, most basic class. Later we'll see that classes can inherit other, more complicated classes. An empty class would look like this:

class ClassName(object):

# class statements go here

Instructions

Define a new class named "Car". For now, since we have to put something inside the class, use the pass keyword.

**Script.py**

class Car(object):

pass

**Output:**

None

**Create an instance of a class**

We can use classes to create new objects, which we say are **instances** of those classes.

Creating a new instance of a class is as easy as saying:

newObject = ClassName()

Instructions

Below your Car class, create a new object named my\_car that is an instance of Car.

**Script.py**

class Car(object):

pass

my\_car = Car()

**Output:**

None

**Class member variables**

Classes can have **member variables** that store information about each class object. We call them *member* variables since they are information that belongs to the class object.

Creating member variables and assigning them initial values is as easy as creating any other variable:

class ClassName(object):

memberVariable = "initialValue"

Instructions

Inside your Car class, replace the pass statement with a new member variable named condition and give it an initial value of the string "new".

**Script.py**

class Car(object):

condition = "new"

my\_car = Car()

**Output:**

None

**Calling class member variables**

Each class object we create has its own set of member variables. Since we've created an object my\_car that is an instance of the Car class, my\_car should already have a member variable named condition. This attribute gets assigned a value as soon as my\_car is created.

Instructions

At the end of your code, use a print statement to display the condition of my\_car.

**Script.py**

class Car(object):

condition = "new"

my\_car = Car()

print my\_car.condition

**Output:**

New

**Initializing a class**

There is a special function named \_\_init\_\_() that gets called whenever we create a new instance of a class. It exists by default, even though we don't see it. However, we can define our own \_\_init\_\_() function inside the class, overwriting the default version. We might want to do this in order to provide more input variables, just like we would with any other function.

The first argument passed to \_\_init\_\_() must always be the keyword self - this is how the object keeps track of itself internally - but we can pass additional variables after that.

In order to assign a variable to the class (creating a member variable), we use **dot notation**. For instance, if we passed newVariable into our class, inside the \_\_init\_\_() function we would say:

self.new\_variable = new\_variable

Instructions

Define the \_\_init\_\_() function of the Car class to take four inputs: self, model, color, and mpg. Assign the last three inputs to member variables of the same name by using the self keyword.

Then, modify the object my\_car to provide the following inputs at initialization:

model = "DeLorean"

color = "silver"

mpg = 88

You don't need to include the self keyword when you create an instance of a class, because self gets added to the beginning of your list of inputs automatically by the class definition.

**Script.py**

class Car(object):

condition = "new"

def \_\_init\_\_(self, model, color, mpg):

self.model = model

self.color = color

self.mpg = mpg

my\_car = Car("Delorean", "silver", 88)

print my\_car.condition

**Output:**

new

**Referring to member variables**

Calling class member variables works the same whether those values are created within the class (like our car's condition) or values are passed into the new object at initialization. We use dot notation to access the member variables of classes since those variables belong to the object.

For instance, if we had created a member variable named new\_variable, a new instance of the class named new\_object could access this variable by saying:

new\_object.new\_variable

Instructions

Now that you've created my\_car print its member variables:

1. First print the model of my\_car. Click "Stuck? Get a hint!" for an example.
2. Then print out the color of my\_car.
3. Then print out the mpg of my\_car.

**Script.py**

class Car(object):

condition = "new"

def \_\_init\_\_(self, model, color, mpg):

self.model = model

self.color = color

self.mpg = mpg

my\_car = Car("DeLorean", "silver", 88)

print my\_car.condition

print my\_car.model

print my\_car.color

print my\_car.mpg

**Output:**

new

DeLorean

silver

88

**Creating class methods**

Besides member variables, classes can also have their own methods. For example:

class Square(object):

def \_\_init\_\_(self, side):

self.side = side

def perimeter(self):

return self.side \* 4

The perimeter() class method is identical to defining any other function, except that it is written inside of the Square class definition.

Just like when we defined \_\_init\_\_(), you need to provide self as the first argument of any class method.

Instructions

1. Inside the Car class, add a method named display\_car() to Car that will reference the Car's member variables to return the string, "This is a [color] [model] with [mpg] MPG." You can use the str() function to turn your mpg into a string when creating the display string.
2. Replace the individual print statements with a single print command that displays the result of calling my\_car.display\_car()

**Script.py**

class Car(object):

condition = "new"

def \_\_init\_\_(self, model, color, mpg):

self.model = model

self.color = color

self.mpg = mpg

def display\_car(self):

return "This is a " + self.color + " " + self.model + " with " + str(self.mpg) + " MPG."

my\_car = Car("DeLorean", "silver", 88)

print my\_car.condition

print my\_car.model

print my\_car.color

print my\_car.mpg

print my\_car.display\_car()

**Output:**

new

DeLorean

silver

88

This is a silver DeLorean with 88 MPG.

**Modifying member variables**

We can modify variables that belong to a class the same way that we initialize those member variables. This can be useful when we want to change the value a variable takes on based on something that happens inside of a class method.

Instructions

1. Inside the Car class, add a method drive\_car() that sets self.condition to the string "used".
2. Remove the call to my\_car.display\_car() and instead print only the condition of your car.
3. Then drive your car by calling the drive\_car() method.
4. Finally, print the condition of your car again to see how its value changes.

**Script.py**

class Car(object):

condition = "new"

def \_\_init\_\_(self, model, color, mpg):

self.model = model

self.color = color

self.mpg = mpg

def display\_car(self):

return "This is a " + self.color + " " + self.model + " with " + str(self.mpg) + " MPG."

def drive\_car(self):

self.condition = "used"

my\_car = Car("DeLorean", "silver", 88)

print my\_car.condition

print my\_car.model

print my\_car.color

print my\_car.mpg

print my\_car.condition

my\_car.drive\_car()

print my\_car.condition

**Output:**

new

DeLorean

silver

88

new

used

**Inheritance**

One of the benefits of classes is that we can create more complicated classes that inherit variables or methods from their **parent classes**. This saves us time and helps us build more complicated objects, since these **child classes** can also include additional variables or methods.

We define a "child" class that inherits all of the variables and functions from its "parent" class like so:

class ChildClass(ParentClass):

# new variables and functions go here

Normally we use object as the parent class because it is the most basic type of class, but by specifying a different class, we can inherit more complicated functionality.

Instructions

Create a class ElectricCar that inherits from Car. Give your new class an \_\_init\_\_() method of that includes a "battery\_type" member variable in addition to the model, color and mpg.

Then, create an electric car named "my\_car" with a "molten salt" battery\_type. Supply values of your choice for the other three inputs (model, color and mpg).

**Script.py**

class Car(object):

condition = "new"

def \_\_init\_\_(self, model, color, mpg):

self.model = model

self.color = color

self.mpg = mpg

def display\_car(self):

return "This is a " + self.color + " " + self.model + " with " + str(self.mpg) + " MPG."

def drive\_car(self):

self.condition = "used"

class ElectricCar(Car):

def \_\_init\_\_(self, model, color, mpg, battery\_type):

self.model = model

self.color = color

self.mpg = mpg

self.battery\_type = battery\_type

#my\_car = Car("DeLorean", "silver", 88)

my\_car = ElectricCar("Prius", "silver", 100, "molten salt")

print my\_car.condition

print my\_car.model

print my\_car.color

print my\_car.mpg

print my\_car.battery\_type

**Output:**

new

Prius

silver

100

molten salt

**Overriding methods**

Since our ElectricCar is a more specialized type of Car, we can give the ElectricCar its own drive\_car() method that has different functionality than the original Car class's.

Instructions

1. Inside ElectricCar add a new method drive\_car() that changes the car's condition to the string "like new".
2. Then, outside of ElectricCar, print the condition of my\_car
3. Next, drive my\_car by calling the drive\_car() function
4. Finally, print the condition of my\_car again

**Script.py**

class Car(object):

condition = "new"

def \_\_init\_\_(self, model, color, mpg):

self.model = model

self.color = color

self.mpg = mpg

def display\_car(self):

return "This is a " + self.color + " " + self.model + " with " + str(self.mpg) + " MPG."

def drive\_car(self):

self.condition = "used"

class ElectricCar(Car):

def \_\_init\_\_(self, model, color, mpg, battery\_type):

self.model = model

self.color = color

self.mpg = mpg

self.battery\_type = battery\_type

def drive\_car(self):

self.condition = "like new"

#my\_car = Car("DeLorean", "silver", 88)

my\_car = ElectricCar("Prius", "silver", 100, "molten salt")

print my\_car.condition

print my\_car.model

print my\_car.color

print my\_car.mpg

my\_car.drive\_car()

print my\_car.condition

print my\_car.battery\_type

**Output:**

new

Prius

silver

100

like new

molten salt

**Building useful classes**

Chances are, you won't be designing Car classes in the real world anytime soon. Usually, classes are most useful for holding and accessing abstract collections of data.

One useful class method to override is the built-in \_\_repr\_\_() method, which is short for *representation*; by providing a return value in this method, we can tell Python how to represent an object of our class (for instance, when using a print statement).

Instructions

1. Define a Point3D class that inherits from object
2. Inside the Point3D class, define an \_\_init\_\_() function that accepts self, x, y, and z, and assigns these numbers to the member variables self.x, self.y, self.z
3. Define a \_\_repr\_\_() method that returns "(%d, %d, %d)" % (self.x, self.y, self.z). This tells Python to represent this object in the following format: (x, y, z).
4. Outside the class definition, create a variable named my\_point containing a new instance of Point3D with x=1, y=2, and z=3.
5. Finally, print my\_point.

**Script.py**

class Point3D(object):

def \_\_init\_\_(self, x, y, z):

self.x = x

self.y = y

self.z = z

def \_\_repr\_\_(self):

return "(%d, %d, %d)" % (self.x, self.y, self.z)

my\_point = Point3D(1, 2, 3)

print my\_point

**Output:**

(1, 2, 3)

**File Input/Output**

**See It to Believe It**

Until now, the Python code you've been writing comes from one source and only goes to one place: you type it in at the keyboard and its results are displayed in the console. But what if you want to read information from a file on your computer, and/or write that information to another file?

This process is called **file I/O** (the "I/O" stands for "input/output"), and Python has a number of built-in functions that handle this for you.

Check out the code in the editor to the right. Note that you now have an extra output.txt tab, which is just an empty text file. That's all about to change!

Instructions

Click Save & Submit Code, then check out the output.txt tab to see Python's file I/O powers in action.

**Script.py**

my\_list = [i\*\*2 for i in range(1,11)]

# Generates a list of squares of the numbers 1 - 10

f = open("output.txt", "w")

for item in my\_list:

f.write(str(item) + "\n")

f.close()

**Output.txt**

1

4

9

16

25

36

49

64

81

100

**The open() Function**

Let's walk through the process of writing to a file one step at a time.

The first code that you saw executed in the previous exercise was this:

f = open("output.txt", "w")

This told Python to open output.txt in "w" mode ("w" stands for "write"). We stored the result of this operation in a file object, f.

Doing this opens the file in write-mode and prepares Python to send data into the file.

Instructions

Create a variable, my\_file, and set it equal to calling the open() function on output.txt. In this case, pass "r+" as a second argument to the function so the file will allow you to read *and* write to it! (See the Hint for details.)

**Script.py**

my\_file = open("output.txt", "r+")

**Output.txt**

None

**Writing**

Good work! Now it's time to write some data to our output.txt file.

We've added the list comprehension from the first exercise to the code in the editor. Our goal in this exercise will be to write each element of that list to output.txt (shown in a new tab above the editor) with each number on its own line.

We can write to a Python file like so:

my\_file.write("Data to be written")

The write() function takes a string argument, so we'll need to do a few things here:

You **must** close the file. You do this simply by calling my\_file.close() (we did this for you in the last exercise). If you don't close your file, Python *won't* write to it properly. From here on out, you gotta close your files!

Instructions

1. Iterate over my\_list to get each value
2. Use my\_file.write() to write each value to output.txt
3. Make sure to call str() on the iterating data so .write() will accept it
4. Make sure to add a newline ("\n") after each element to ensure each will appear on its own line.
5. Use my\_file.close() to close the file when you're done.

Script.py

my\_list = [i\*\*2 for i in range(1,11)]

my\_file = open("output.txt", "r+")

# Add your code below!

for item in my\_list:

my\_file.write(str(item)+"\n")

my\_file.close()

**Output.txt**

1

4

9

16

25

36

49

64

81

100

**Reading**

Excellent! You're a pro.

Finally, we want to know how to read from our output.txt file. As you might expect, we do this with the read() function, like so:

print my\_file.read()

Instructions

1. Declare a variable, my\_file, and set it equal to the file object returned by calling open() with both "output.txt" and "r".
2. Next, print the result of using .read() on my\_file, like the example above.
3. Make sure to .close() your file when you're done with it! All kinds of doom will happen if you don't.

**Script.py**

my\_file = open("outout.txt", "r")

print my\_file.read()

my\_file.close()

**Output.txt**

None

**Reading Between the Lines**

What if we want to read from a file line by line, rather than pulling the entire file in at once. Thankfully, Python includes a readline() function that does exactly that.

If you open a file and call .readline() on the file object, you'll get the first line of the file; subsequent calls to .readline() will return successive lines.

Instructions

1. Declare a new variable my\_file and store the result of calling open() on the "text.txt" file in "r"ead-only mode.
2. On three separate lines, print out the result of calling my\_file.readline(). See how it gets the next line each time?
3. Don't forget to close() your file when you're done with it!)

**Script.py**

my\_file = open("text.txt", "r")

print my\_file.readline()

print my\_file.readline()

print my\_file.readline()

my\_file.close()

**text.txt**

I'm the first line of the file!

I'm the second line.

Third line here, boss.

**PSA: Buffering Data**

We keep telling you that you always need to close your files after you're done writing to them. Here's why!

During the I/O process, data is **buffered**: this means that it is held in a temporary location before being written to the file.

Python doesn't **flush the buffer**—that is, write data to the file—until it's sure you're done writing. One way to do this is to close the file. If you write to a file without closing, the data won't make it to the target file.

Instructions

Check out our extremely bad code in the editor. Click Save & Submit Code—you'll note that our read\_file.read() didn't read any data back! (The text still appears in text.txt, though, because we closed the file behind the scenes for you. Safety first!)

1. Add a write\_file.close() call on [line 9](javascript:void(0)).
2. Add a read\_file.close() on [line 13](javascript:void(0)).
3. Run the code again.
4. This time, you'll see the data come through!

Script.py

# Open the file for reading

read\_file = open("text.txt", "r")

# Use a second file handler to open the file for writing

write\_file = open("text.txt", "w")

# Write to the file

write\_file.write("Not closing files is VERY BAD.")

write\_file.close()

# Try to read from the file

print read\_file.read()

read\_file.close()

**text.txt**

Not closing files is VERY BAD.

**The 'with' and 'as' Keywords**

Programming is all about getting the computer to do the work. Is there a way to get Python to automatically close our files for us?

Of course there is. This is Python.

You may not know this, but file objects contain a special pair of built-in methods: \_\_enter\_\_() and \_\_exit\_\_(). The details aren't important, but what *is* important is that when a file object's \_\_exit\_\_() method is invoked, it automatically closes the file. How do we invoke this method? With with and as.

The syntax looks like this:

with open("file", "mode") as variable:

# Read or write to the file

Instructions

Check out the example in the editor. Note that we don't explicitly close() our file, and remember that if we don't close a file, our data will get stuck in the buffer. Click Save & Submit Code and check out text.txt to see the results.

**Script.py**

with open("text.txt", "w") as textfile:

textfile.write("Success!")

**text.txt**

Success!

**Try It Yourself**

It worked! Our Python program successfully wrote to text.txt.

Instructions

Now you try: write any data you like to the text.txt file using with...as. Give your file object the usual name: my\_file.

**Script.py**

with open("text.txt", "w") as my\_file:

my\_file.write("Here I wrote something.")

**text.txt**

Here I wrote something.

**Case Closed?**

Finally, we'll want a way to test whether a file we've opened is closed. Sometimes we'll have a lot of file objects open, and if we're not careful, they won't all be closed. How can we test this?

f = open("bg.txt")

f.closed

# False

f.close()

f.closed

# True

Python file objects have a closed attribute which is True when the file is closed and False otherwise.

By checking file\_object.closed, we'll know whether our file is closed and can call close() on it if it's still open.

Instructions

Below your with...as code, do two things:

1. Check if the file is *not* .closed.
2. If that's the case, call .close() on it.
3. (You don't need an else here, since your if statement should do nothing if .closed is True.)
4. After your if statement, print out the value of my\_file.closed to make sure your file is really closed.

**Script.py**

with open("text.txt", "w") as my\_file:

my\_file.write("Here I wrote something.")

if my\_file is not my\_file.closed:

my\_file.close()

print my\_file.closed

**text.txt**

True