**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