# Week 2 Notes

# Welcome to Python Functions!

This lesson will teach you how to create a reusable chunk of code to solve a specific purpose. Programmers use functions constantly to keep their code clean and readable. Knowing how to create and use functions will be essential as you continue your journey through the world of code.

Introduction to Functions

# What is a Function?

Let's imagine that we are creating a program that greets customers as they enter a grocery store. We want a big screen at the entrance of the store to say:

Welcome to Engrossing Grocer's.

Our special is mandarin oranges.

Have fun shopping!

We have learned to use print statements for this purpose:

print("Welcome to Engrossing Grocer's.")

print("Our special is mandarin oranges.")

print("Have fun shopping!")

Every time a customer enters, we call these three lines of code. Even if only 3 or 4 customers come in, that's a lot of lines of code required.

In Python, we can make this process easier by assigning these lines of code to a function. We'll name this function greet\_customer. In order to call a function, we use the syntax function\_name(). The parentheses are important! They make the code inside the function run. In this example, the function call looks like:

greet\_customer()

Every time we call greet\_customer, we would see:

Welcome to Engrossing Grocer's.

Our special is mandarin oranges.

Have fun shopping!

Having this functionality inside greet\_customer is better form, because we have isolated this behavior from the rest of our code. Once we determine that greet\_customer works the way we want, we can reuse it anywhere and be confident that it greets, without having to look at the implementation. We can get the same output, with less repeated code. Repeated code is generally more error prone and harder to understand, so it's a good goal to reduce the amount of it.

**1.**

In **script.py**, we have made a function for you called sing\_song. Call this function once to see what it prints out.

def sing\_song():

print("You may say I'm a dreamer")

print("But I'm not the only one")

print("I hope some day you'll join us")

print("And the world will be as one")

sing\_song()

**2.**

Now call sing\_song three times total.

def sing\_song():

print("You may say I'm a dreamer")

print("But I'm not the only one")

print("I hope some day you'll join us")

print("And the world will be as one")

sing\_song()

sing\_song()

sing\_song()

# Write a Function

We have seen the value of simple functions for modularizing code. Now we need to understand how to write a function. To write a function, you must have a heading and an indented block of code. The heading starts with the keyword def and the name of the function, followed by parentheses, and a colon. The indented block of code performs some sort of operation. This syntax looks like:

def function\_name():

some code

For our greet\_customer example, the function definition looks like:

def greet\_customer():

print("Welcome to Engrossing Grocer's.")

print("Our special is mandarin oranges.")

print("Have fun shopping!")

The keyword def tells Python that we are defining a function. This function is called greet\_customer. Everything that is indented after the : is what is run when greet\_customer is called. So every time we call greet\_customer, the three print statements run.

**1.**

Write a function called loading\_screen that prints "This page is loading…" to the console.

**2.**

Call loading\_screen().

Answer:

def loading\_screen():

print("This page is loading...")

loading\_screen()

# Whitespace

Consider this function:

def greet\_customer():

print("Welcome to Engrossing Grocer's.")

print("Our special is mandarin oranges.")

print("Have fun shopping!")

The three print statements are all executed together when greet\_customer is called. This is because they have the same level of indentation. In Python, the amount of whitespace tells the computer what is part of a function and what is not part of that function. If we wanted to write another line outside of greet\_customer, we would have to unindent the new line:

def greet\_customer():

print("Welcome to Engrossing Grocer's.")

print("Our special is mandarin oranges.")

print("Have fun shopping!")

print("Cleanup on Aisle 6")

When we call greet\_customer, the message"Cleanup on Aisle 6" is not printed, as it is not part of the function.

Here at Codecademy, we use 2 spaces for our default indentation. Anything other than that will throw an error when you try to run the program. Many other platforms use 4 spaces. Some people even use tabs! These are all fine. What is important is being consistent throughout the project.

Instructions

1. Run **script.py**. Look at what is printed out!

def about\_this\_computer():

print("This computer is running on version Everest Puma")

print("This is your desktop")

about\_this\_computer()

about\_this\_computer()

Output =

This computer is running on version Everest Puma

This is your desktop

This computer is running on version Everest Puma

This is your desktop

1. Remove the indent on the second print statement. Run the file. Now what's printed?

def about\_this\_computer():

print("This computer is running on version Everest Puma")

print("This is your desktop")

about\_this\_computer()

about\_this\_computer()

Output:

This is your desktop

This computer is running on version Everest Puma

This computer is running on version Everest Puma

# Parameters

Let's return to Engrossing Grocer's. The special of the day will not always be mandarin oranges, it will change every day. What if we wanted to call these three print statements again, except with a variable special? We can use parameters, which are variables that you can pass into the function when you call it.

def greet\_customer(special\_item):

print("Welcome to Engrossing Grocer's.")

print("Our special is " + special\_item + ".")

print("Have fun shopping!")

In the definition heading for greet\_customer, the special\_item is referred to as a formal parameter. This variable name is a placeholder for the name of the item that is the grocery's special today. Now, when we call greet\_customer, we have to provide a special\_item:

greet\_customer("peanut butter")

That item will get printed out in the second print statement:

Welcome to Engrossing Grocer's.

Our special is peanut butter.

Have fun shopping!

The value between the parentheses when we call the function (in this case, "peanut butter") is referred to as an argument of the function call. The argument is the information that is to be used in the execution of the function. When we then call the function, Python assigns the formal parameter name special\_item with the actual parameter data, "peanut\_butter". In other words, it is as if this line was included at the top of the function:

special\_item = "peanut butter"

Every time we call greet\_customer with a different value between the parentheses, special\_item is assigned to hold that value.

Instructions

1. The function mult\_two\_add\_three prints a number multiplied by 2 and added to 3. As it is written right now, the number that it operates on is always 5. Call the function and see what it prints to the console.

2. Now, modify the function definition so that it has a parameter called number. Then delete the number = 5 assignment on the first line of the function.

Pass the number 1 into your function call.

3. Call the function with the value 5 as the argument.

4. Call the function with the value -1 as the argument.

5. Call the function with the value 0 as the argument.

Solultion:

def mult\_two\_add\_three(number):

print(number\*2 + 3)

mult\_two\_add\_three(5)

mult\_two\_add\_three(-1)

mult\_two\_add\_three(0)

# Multiple Parameters

Our grocery greeting system has gotten popular, and now other supermarkets want to use it. As such, we want to be able to modify both the special item and the name of the grocery store in a greeting like this:

Welcome to [grocery store].

Our special is [special item].

Have fun shopping!

We can make a function take more than one parameter by using commas:

def greet\_customer(grocery\_store, special\_item):

print("Welcome to "+ grocery\_store + ".")

print("Our special is " + special\_item + ".")

print("Have fun shopping!")

The variables grocery\_store and special\_item must now both be provided to the function upon calling it:

greet\_customer("Stu's Supplies", "papayas")

which will print:

Welcome to Stu's Supplies.

Our special is papayas.

Have fun shopping!

Instructions

**1.**

The function mult\_two\_add\_three takes a number, multiplies it by two and adds three. We want to make this more flexible. First, change the name of the function to mult\_x\_add\_y.

**2.**

Now, add x and y as parameters of the function, after number.

**3.**

Inside the function, replace 2 in the print statement with x, and replace 3 in the print statement with y.

**4.**

Call the function with these values:

number: 5

x: 2

y: 3

Solution:

def mult\_x\_add\_y(number,x,y):

print(number\*x + y)

mult\_x\_add\_y(5,2,3)

**5.**

Call the function with these values:

number: 1

x: 3

y: 1

Solution:

def mult\_x\_add\_y(number,x,y):

print(number\*x + y)

mult\_x\_add\_y(1,3,1)

# Keyword Arguments

In our greet\_customer function from the last exercise, we had two arguments:

def greet\_customer(grocery\_store, special\_item):

print("Welcome to "+ grocery\_store + ".")

print("Our special is " + special\_item + ".")

print("Have fun shopping!")

Whichever value is put into greet\_customer first is assigned to grocery\_store, and whichever value is put in second is assigned to special\_item. These are called positional arguments, because their assignments depend on their positions in the function call. We can also use keyword arguments, which are assigned explicitly in the function definition:

def greet\_customer(special\_item, grocery\_store="Engrossing Grocer's"):

print("Welcome to "+ grocery\_store + ".")

print("Our special is " + special\_item + ".")

print("Have fun shopping!")

In this case, grocery\_store is our keyword argument. We have set it to a default of "Engrossing Grocer's". If we call the function with only one argument, the value of "Engrossing Grocer's" is used for grocery\_store:

greet\_customer("bananas")

Welcome to Engrossing Grocer's.

Our special is bananas.

Have fun shopping!

But we can also supply a value for grocery\_store:

greet\_customer("chips and salsa", grocery\_store="Stu's Staples")

Welcome to Stu's Staples.

Our special is chips and salsa.

Have fun shopping!

Instructions

1. Define a function called create\_spreadsheet that takes one argument, title, and only prints the string "Creating a spreadsheet called [TITLE]", where [TITLE] is replaced with the value of title.
   1. Solution:

def create\_spreadsheet(title):

print("Creating a spreadsheet called "+title+" where "+title+" is replaced with the value of "+title+"" ".")

create\_spreadsheet("Downloads")

1. Call create\_spreadsheet, with a title of "Downloads".
   1. Solution:

create\_spreadsheet("Downloads")

1. Add a keyword argument row\_count to the function definition. Set the default value to be 1000.
   1. Solution:

def create\_spreadsheet(title, row\_count = 1000):

1. Change the print statement in the function to print "Creating a spreadsheet called [TITLE] with [ROW COUNT] rows", where [ROW COUNT] is replaced with the value of row\_count.
   1. Remember, to concatenate a number to a string object, you'll first have to cast row\_count to a string using str().
   2. Solution:

def create\_spreadsheet(title, row\_count = 1000):

print("Creating a spreadsheet called "+title+" with "+str(row\_count)+" rows.")

1. Call create\_spreadsheet with title set to "Applications" and row\_count set to 10.
   1. Solution:

def create\_spreadsheet(title, row\_count = 1000):

print("Creating a spreadsheet called "+title+" with "+str(row\_count)+" rows.")

create\_spreadsheet("Applications", 10)

# Returns

So far, we have only seen functions that print out some result to the console. Functions can also return a value to the user so that this value can be modified or used later. When there is a result from a function that can be stored in a variable, it is called a returned function value. We use the keyword return to do this.

Here's an example of a function divide\_by\_four that takes an integer argument, divides it by four, and returns the result:

def divide\_by\_four(input\_number):

return input\_number/4

The program that calls divide\_by\_four can then use the result later:

result = divide\_by\_four(16)

# result now holds 4

print("16 divided by 4 is " + str(result) + "!")

result2 = divide\_by\_four(result)

print(str(result) + " divided by 4 is " + str(result2) + "!")

This would print out:

16 divided by 4 is 4!

4 divided by 4 is 1!

In this example, we returned a number, but we could also return a String:

def create\_special\_string(special\_item):

return "Our special is " + special\_item + "."

special\_string = create\_special\_string("banana yogurt")

print(special\_string)

Our special is banana yogurt.

Instructions

1. The function calculate\_age in **script.py** creates a variable called age that is the difference between the current year, and a birth year, both of which are inputs of the function. Add a line to return age.
   1. Remember that whitespace is important in Python, so make sure that your return statement has the same indentation level as the line with the definition of age.
      1. Solution:

def calculate\_age(current\_year, birth\_year):

age = current\_year - birth\_year

return age

1. Outside of the function, call calculate\_age with values 2018 (current\_year) and 1993 (birth\_year) and save the value to a variable called my\_age.
   1. Solution:

def calculate\_age(current\_year, birth\_year):

age = current\_year - birth\_year

return age

my\_age = calculate\_age(2018, 1993)

1. Call calculate\_age with values 2018 (current\_year) and 1953 (birth\_year) and save the value to a variable called dads\_age.
   1. Solution:

def calculate\_age(current\_year, birth\_year):

age = current\_year - birth\_year

return age

my\_age = calculate\_age(2018, 1993)

dads\_age = calculate\_age(2018, 1953)

1. Print the string "I am X years old and my dad is Y years old" to the console, with my\_age where the X is and dads\_age where the Y is.
   1. Solution:

def calculate\_age(current\_year, birth\_year):

age = current\_year - birth\_year

return age

my\_age = calculate\_age(2018, 1993)

dads\_age = calculate\_age(2018, 1953)

print("I am "+str(my\_age)+" years old and my dad is "+str(dads\_age)+" years old")

Hint: for printing out sentences, don't forget to convert the age values to strings using str()!

# Multiple Return Values

Sometimes we may want to return more than one value from a function. We can return several values by separating them with a comma:

def square\_point(x\_value, y\_value):

x\_2 = x\_value \* x\_value

y\_2 = y\_value \* y\_value

return x\_2, y\_2

This function takes in an x value and a y value, and returns them both, squared. We can get those values by assigning them both to variables when we call the function:

x\_squared, y\_squared = square\_point(1, 3)

print(x\_squared)

print(y\_squared)

This will print:

1

9

1. Write a function called get\_boundaries() that takes in two parameters, a number called target and a number called margin.

It should create two variables:

1. low\_limit: margin subtracted from target
2. high\_limit: margin added to target
3. a subtracted from b is equivalent to b – a
   * Solution:

def get\_boundaries(target, margin):

low\_limit = target - margin

high\_limit = target + margin

1. Return both low\_limit and high\_limit from the function.
   1. Solution:

def get\_boundaries(target, margin):

low\_limit = target - margin

high\_limit = target + margin

return low\_limit, high\_limit

1. Call the function on the target 100 with a margin of 20. Save the returned values to variables called low and high.
   1. Solution:

def get\_boundaries(target, margin):

low\_limit = target - margin

high\_limit = target + margin

return low\_limit, high\_limit

low, high = get\_boundaries(100, 20)

1. Print out a string that says: Low limit: {low}, high limit: {high} with the values of low and high that you got from the get\_boundaries() function.
   1. Solution:

def get\_boundaries(target, margin):

low\_limit = target - margin

high\_limit = target + margin

return low\_limit, high\_limit

low, high = get\_boundaries(100, 20)

print("Low limit: "+str(low)+", high limit: "+str(high)+ ".")

Hint: To print the sentence, don't forget to convert the age values to strings using str()!

# Scope

Let's say we have our function from the last exercise that creates a string about a special item:

def create\_special\_string(special\_item):

return "Our special is " + special\_item + "."

What if we wanted to access the variable special\_item outside of the function? Could we use it?

def create\_special\_string(special\_item):

return "Our special is " + special\_item + "."

print("I don't like " + special\_item)

If we try to run this code, we will get a NameError, telling us that 'special\_item' is not defined. The variable special\_item has only been defined inside the space of a function, so it does not exist outside the function. We call the part of a program where special\_item can be accessed its scope. The scope of special\_item is only the create\_special\_string function.

Variables defined outside the scope of a function may be accessible inside the body of the function:

header\_string = "Our special is "

def create\_special\_string(special\_item):

return header\_string + special\_item + "."

print(create\_special\_string("grapes"))

There is no error here. header\_string can be used inside the create\_special\_string function because the scope of header\_string is the whole file. This file would produce:

Our special is grapes.

Instructions:

**1.** Outside of the function calculate\_age, try to print current\_year. Does it work?

**2.** It doesn't! That's because we haven't defined current\_year outside the scope of the function.

Run the function calculate\_age(2018, 1970).

This won't produce any output so don't worry if you don't see any.

**3.** Try to print current\_year again outside of the function. Did it work this time?

**4.**No, current\_year still only exists in the context of the function. Delete your last print statement.

What about age? Outside of calculate\_age, try to print age. Does it work?

1. No! Even though we return age at the end of the function, the variable age still only exists within the context of the function.
   1. Let's try something different. Remove the parameter current\_year so that it is no longer a parameter of calculate\_age.
   2. Define current\_year as a variable OUTSIDE of the function and give it the value 2018. Now, every time calculate\_age is called, it should only take birth\_year.
2. Try to print current\_year one last time. Does it work finally?

**7.** Hooray! Now we have current\_year globally defined. Great work!

Let's make sure our function still works: print the value of calculate\_age with 1970 as the parameter.

Solution:

def calculate\_age(birth\_year):

age = current\_year - birth\_year

return age

#print(current\_year)

#calculate\_age(2018,1970)

#print(age)

current\_year=2018

print(current\_year)

print(calculate\_age(1970))

# Review

Great! So far you have learned:

* how to write a function
* how to give a function inputs
* how to return values from a function
* what scope means

Let's practice these concepts again so that you won't forget them!

**1.** Define a function called repeat\_stuff that takes in two inputs, stuff, and num\_repeats.

We will want to make this function print a string with stuff repeated num\_repeats amount of times. For now, only put an empty print statement inside the function.

To define a function, use the syntax:

def your\_function(your\_input1, your\_input2):

#some piece of code like print()

1. Outside of the function, call repeat\_stuff.

You can use the value "Row " for stuff and 3 for num\_repeats.

To call a function, use parentheses:

your\_function()

1. Change the print statement inside repeat\_stuff to a return statement instead.

It should return stuff\*num\_repeats.

**Note:** Multiplying a string just makes a new string with the old one repeated! For example:

"na"\*6

results in the string "nananananana".

return does not use parentheses:

def your\_function(your\_input1, your\_input2):

return some\_value

1. Give the parameter num\_repeats a default value of 10.

In the function definition, you can assign default values to parameters using this syntax:

def your\_function(your\_input1, your\_input2=default\_value):

...

Now, if your\_input2 is not supplied, it will take the value default\_value.

1. Add repeat\_stuff("Row ", 3) to the string "Your Boat" and save the result to a variable called lyrics.

To add two strings and save them to a variable, we can use this syntax:

big\_string = small\_string + another\_small\_string

1. Create a variable called song and assign it the value of repeat\_stuff called with the singular input lyrics.

You can call a function with one input using the syntax:

your\_function(your\_input)

1. Print song.

Good job!

def repeat\_stuff(stuff, num\_repeats=10):

return stuff\*num\_repeats

repeat\_stuff("Row ",3)

lyrics = repeat\_stuff("Row ", 3) + "Your Boat"

song = repeat\_stuff(lyrics)

print(song)

# Getting ready for Physics Class

You are a physics teacher preparing for the upcoming semester. You want to provide your students with some functions that will help them calculate some fundamental physical properties.

Mark the tasks as complete by checking them off

### Turn up the Temperature

1.

Write a function called f\_to\_c that takes an input f\_temp, a temperature in Fahrenheit, and converts it to c\_temp, that temperature in Celsius.

It should then return c\_temp.

The equation you should use is:

Temp (C) = (Temp (F) - 32) \* 5/9

To define a function with an input, use this syntax:

def your\_function(your\_input):

... #do something with the input

return some\_final\_value

2.

Let's test your function with a value of 100 Fahrenheit. Define a variable f100\_in\_celsius and set it equal to the value of f\_to\_c with 100 as an input.

To call a function use the syntax:

returned\_value = your\_function(your\_input)

3.

Write a function called c\_to\_f that takes an input c\_temp, a temperature in Celsius, and converts it to f\_temp, that temperature in Fahrenheit.

It should then return f\_temp.

The equation you should use is:

Temp (F) = Temp (C) \* (9/5) + 32

4.

Let's test your function with a value of 0 Celsius. Define a variable c0\_in\_fahrenheit and set it equal to the value of c\_to\_f with 0 as an input.

You can print() the value of c0\_in\_fahrenheit to check that it equals 32.0.

### Use the Force

5.

Define a function called get\_force that takes in mass and acceleration. It should return mass multiplied by acceleration.

6.

Test get\_force by calling it with the variables train\_mass and train\_acceleration. Save the result to a variable called train\_force and print it out.

train\_force should equal 226800.

7.

Print the string "The GE train supplies X Newtons of force.", with X replaced by train\_force.

8.

Define a function called get\_energy that takes in mass and c.

c is a constant that is usually set to the speed of light, which is roughly 3 x 10^8. Set c to have a default value of 3\*10\*\*8.

get\_energy should return mass multiplied by c squared.

9.

Test get\_energy by using it on bomb\_mass, with the default value of c. Save the result to a variable called bomb\_energy.

Bombs have a lot of energy: 90000000000000000 of it.

10.

Print the string "A 1kg bomb supplies X Joules.", with X replaced by bomb\_energy.

### Do the Work

11.

Define a final function called get\_work that takes in mass, acceleration, and distance.

Work is defined as force multiplied by distance. First, get the force using get\_force, then multiply that by distance. Return the result.

To call get\_force, use mass and acceleration

12.

Test get\_work by using it on train\_mass, train\_acceleration, and train\_distance. Save the result to a variable called train\_work.

13.

Print the string "The GE train does X Joules of work over Y meters.", with X replaced with train\_work and Y replaced with train\_distance.

Remember to cast train\_work and train\_distance to strings using str() before concatenating.

The GE train does 22680000 Joules of work over 100, by the way.

def get\_force(mass, acceleration):

force = mass\*acceleration

return force

def get\_energy(mass, c=3\*10\*\*8):

energy = mass\*c\*\*2

return energy

def f\_to\_c(f\_temp):

c\_temp = (f\_temp - 32) \* (5/9)

return c\_temp

def c\_to\_f(c\_temp):

f\_temp = c\_temp \* (9/5) + 32

return c\_temp

def get\_work(mass, acceleration, distance):

force = get\_force(mass, acceleration)

return force\*distance

f100\_in\_celsius = f\_to\_c(100)

c0\_in\_fahrenheit = c\_to\_f(0)

print("100F is " + str(f100\_in\_celsius) + " in Celsius")

print("0C is " + str(c0\_in\_fahrenheit) + " in Fahrenheit")

train\_mass = 22680

train\_acceleration = 10

train\_distance = 100

train\_force = get\_force(train\_mass, train\_acceleration)

train\_work = get\_work(train\_mass, train\_acceleration, train\_distance)

bomb\_mass = 1

bomb\_energy = get\_energy(bomb\_mass)

print("The GE train supplies " + str(train\_force) + " Newtons of force.")

print("The GE train does " + str(train\_force) + " Joules of work over "+str(train\_distance)+" meters.")

print("A 1kg bomb supplies " + str(bomb\_energy) + " Joules.")

**Introduction**

This lesson will help you review Python functions by providing some challenge exercises.

As a refresher, function syntax looks like this:

def some\_function(some\_input1, some\_input2):

… do something with the inputs …

return output

For example, a function that takes in a number and returns that number multiplied by 4 would look like:

def mult\_by\_four(number):

multiplied = number \* 4

return multiplied

And this would produce output like:

>>> mult\_by\_four(10)

40

>>> mult\_by\_four(0)

0

>>> mult\_by\_four(-1)

-4

Instructions

When you're ready to do this series of short function challenges, continue on to the rest of the lesson!

**Average**

average()

1. Write a function named average that has two parameters named num1 and num2. The function should return the average of these two numbers.
2. Solution:

def average(num1, num2):

return (num1+num2)/2

# Uncomment these function calls to test your average function:

print(average(1, 100))

# The average of 1 and 100 is 50.5

print(average(1, -1))

# The average of 1 and -1 is 0

# Tenth Power

tenth\_power()

Instructions

1. Write a function named tenth\_power that has one parameter named num. The function should return num raised to the 10th power.
2. Solution:

# Write your tenth\_power function here:

def tenth\_power(num):

return num\*\*10

# Uncomment these function calls to test your tenth\_power function:

print(tenth\_power(1))

# 1 to the 10th power is 1

print(tenth\_power(0))

# 0 to the 10th power is 0

print(tenth\_power(2))

# 2 to the 10th power is 1024

# Bond

introduction()

Instructions

1. Write a function named introduction that has two parameters named first\_name and last\_name. The function should return the last\_name, followed by a comma, a space, first\_name another space, and finally last\_name.
2. Use the + operator to concatenate strings together. Don't forget to add the comma or spaces!
   1. "last\_name" + ", " + "first\_name" is a good starting point. Just add a final space and last\_name.
3. Solution:

# Write your introduction function here:

def introduction(first\_name, last\_name):

return last\_name + ", " + first\_name + " " + last\_name

# Uncomment these function calls to test your introduction function:

print(introduction("James", "Bond"))

# should print Bond, James Bond

print(introduction("Maya", "Angelou"))

# should print Angelou, Maya Angelou

# Square Root

square\_root()

Instructions

1. Write a function named square\_root that has one parameter named num. Use exponents (\*\*) to return the square root of num.
2. Hint: Raising a number to 0.5 will result in the square root of that number.
3. Solution:

# Write your square\_root function here:

def square\_root(num):

return num\*\*0.5

# Uncomment these function calls to test your square\_root function:

print(square\_root(16))

# should print 4

print(square\_root(100))

# should print 10

# Tip

tip()

* Create a function called tip that has two parameters named total and percentage. This function should return the amount you should tip given a total and the percentage you want to tip.
* Hint: First divide percentage by 100 and multiply the result by total
* Solution:

# Write your tip function here:

def tip(total, percentage):

return percentage/100\*total

# Uncomment these function calls to test your tip function:

print(tip(10, 25))

# should print 2.5

print(tip(0, 100))

# should print 0.0

# Win Percentage

win\_percentage()

Instructions

* Create a function called win\_percentage that takes two parameters named wins and losses. This function should return out the total percentage of games won by a team based on these two numbers.
* Hint: You can calculate the win percentage by dividing wins by the sum of wins and losses, and multiplying by 100.
* Solutions:

# Write your win\_percentage function here:

def win\_percentage(wins, losses):

total\_games = wins+losses

ratio\_won = wins/total\_games

return ratio\_won\*100

# Uncomment these function calls to test your win\_percentage function:

print(win\_percentage(5, 5))

# should print 50

print(win\_percentage(10, 0))

# should print 100

# First Three Multiple

Instructions

* Write a function named first\_three\_multiples that has one parameter named num.
* This function should print the first three multiples of num.
* Then, it should return the third multiple.
* For example, first\_three\_multiples(7) should print 7, 14, and 21 on three different lines, and return 21.
* Hint: Inside your function, use three separate print statements.
* Solution:
  + # Write your first\_three\_multiples function here:
  + def first\_three\_multiples(num):
  + print(num\*1)
  + print(num\*2)
  + print(num\*3)
  + return num\*3
  + # Uncomment these function calls to test your first\_three\_multiples function:
  + first\_three\_multiples(10)
  + # should print 10, 20, 30, and return 30
  + first\_three\_multiples(0)
  + # should print 0, 0, 0, and return 0

# Dog Years

dog\_years()

* Some say that every one year of a human's life is equivalent to seven years of a dog's life. Write a function named dog\_years that has two parameters named name and age. The function should compute the age in dog years and return the following string:
  + "{name}, you are {age} years old in dog years"
* Test this function with your name and your age!
* Hint: After multiplying age by 7, concatenate the result with correct string using +. Don't forget the comma after the name!
* Solution:
  + # Write your dog\_years function here:
  + def dog\_years(name, age):
  + return name+", you are "+str(age\*7)+" years old in dog years"
  + # Uncomment these function calls to test your dog\_years function:
  + print(dog\_years("Lola", 16))
  + # should print "Lola, you are 112 years old in dog years"
  + print(dog\_years("Baby", 0))
  + # should print "Baby, you are 0 years old in dog years"

# Remainder

remainder()

Instructions

* Write a function named remainder that has two parameters named num1 and num2. The function should return the remainder of twice num1 divided by half of num2.
* Hint: Modulo (%) has the same precedence as multiplication (\*) and division (/). Make sure to do the multiplication and division before the modulo by putting using parentheses like this: ( \_\_\_\_ \* \_\_\_\_) % (\_\_\_\_ / \_\_\_\_)
* Solution:
  + # Write your remainder function here:
  + def remainder(num1, num2):
  + return (2\*num1)%(num2/2)
  + # Uncomment these function calls to test your remainder function:
  + print(remainder(15, 14))
  + # should print 2
  + print(remainder(9, 6))
  + # should print 0

# All Operations

all\_operations()

Instructions

* Create a function named lots\_of\_math. This function should have four parameters named a, b, c, and d. The function should print 4 lines.
* First, the sum of a and b.
* Second, d subtracted from c.
* Third, the first number printed, multiplied by the second number printed.
* Finally, it should return the third number printed mod a.
* Hint: You could create new variables, so you don't have to repeat code. For example,
  + first = a + b
  + second = c – d
  + You could then use your variables first and second when computing the third number.
* Solution:
  + # Write your lots\_of\_math function here:
  + def lots\_of\_math(a, b, c, d):
  + first = a+b
  + second = c-d
  + third = first\*second
  + fourth = third%a
  + print(first)
  + print(second)
  + print(third)
  + return fourth
  + # Uncomment these function calls to test your lots\_of\_math function:
  + print(lots\_of\_math(1, 2, 3, 4))
  + # should print 3, -1, -3, 0
  + print(lots\_of\_math(1, 1, 1, 1))
  + # should print 2, 0, 0, 0

# Control Flow

Welcome to Learn Python: Control Flow!

In this lesson, you will learn how to build control flow into your python code by including if, else, and elif statements as well as try and except statements. You will also learn all you need to know about boolean variables and logical operators.

# Control Flow: An Introduction

Imagine waking up in the morning.

You wake up and think,

"Ugh, is it a weekday?"

If so, you have to get up and get dressed and get ready for work or school. If not, you can sleep in a bit longer and catch a couple extra Z's. But alas, it is a weekday, so you are up and dressed and you go to look outside, "What's the weather like? Do I need an umbrella?"

These questions and decisions control the flow of your morning, each step and result is a product of the conditions of the day and your surroundings. Your computer, just like you, goes through a similar flow every time it executes code. A program will run (wake up) and start moving through its checklists, is this condition met, is that condition met, okay let's execute this code and return that value.

This is the Control Flow of your program. In Python, your script will execute from the top down, until there is nothing left to run. It is your job to include gateways, known as conditional statements, to tell the computer when it should execute certain blocks of code. If these conditions are met, then run this function.

Over the course of this lesson, you will learn how to build conditional statements using boolean expressions and manage the control flow in your code.

Instructions

Click **Next** to proceed to the next exercise.

# Boolean Expressions

In order to build control flow into our program, we want to be able to check if something is true or not. A boolean expression is a statement that can either be True or False.

Let's go back to the 'waking up' example. The first question, "Is today a weekday?" can be written as a boolean expression:

Today is a weekday.

This expression can be True if today is Tuesday, or it can be False if today is Saturday. There are no other options.

Consider the phrase:

Friday is the best day of the week.

Is this a boolean expression?

No, this statement is an opinion and is not objectively True or False. Someone else might say that "Wednesday is the best weekday," and their statement would be no less True or False than the one above.

How about the phrase:

Sunday starts with the letter 'C'.

Is this a boolean expression?

Yes! This expression can only be True or False, which makes it a boolean expression. Even though the statement itself is false (Sunday starts with the letter 'C'), it is still a boolean expression.

**1.**

Determine if the following statements are boolean expressions or not. If they are, set the matching variable to the right to "Yes" and if not set the variable to "No". Here's an example of what to do:

Example statement:

My dog is the cutest dog in the world.

This is an opinion and not a boolean expression, so you would set example\_statement to "No" in the editor to the right. Okay, now it's your turn:

Statement one:

Dogs are mammals. – Yes. Dogs are mammals is a true

Statement two:

My dog is named Pavel. Yes, The dogs name is Pavel is a true statement

Statement three:

Dogs make the best pets. No, this is an opinion

Statement four:

Cats are just female dogs. Yes, This is a false so it makes it a true Boolean expression.

# Relational Operators: Equals and Not Equals

Now that we understand what boolean expressions are, let's learn to create them in Python. We can create a boolean expression by using relational operators.

Relational operators compare two items and return either True or False. For this reason, you will sometimes hear them called comparators.

The two boolean operators we'll cover first are:

* Equals: ==
* Not equals: !=

These operators compare two items and return True or False if they are equal or not.

We can create boolean expressions by comparing two values using these operators:

>>> 1 == 1

True

>>> 2 != 4

True

>>> 3 == 5

False

>>> '7' == 7

False

Each of these is an example of a boolean expression. >>> is the prompt when you run Python in your terminal, which you can then use to evaluate simple expressions, such as these

Why is the last statement false? The '' marks in '7' make it a string, which is different from the integer value 7, so they are not equal. When using relational operators it is important to always be mindful of type.

**Instructions**

Determine if the following boolean expressions are True or False. Input your answer as True or False in the appropriate variable to the right.

Statement one:

(5 \* 2) - 1 == 8 + 1 = True

Statement two:

13 - 6 != (3 \* 2) + 1 = False

Statement three:

3 \* (2 - 1) == 4 – 1 = True

Simplify each side of the expressions using basic order of operations, then determine if the expression is True or False.

# Boolean Variables

Before we go any further, let's talk a little bit about True and False. You may notice that when you type them in the code editor (with uppercase T and F), they appear in a different color than variables or strings. This is because True and False are their own special type: bool.

True and False are the only bool types, and any variable that is assigned one of these values is called a boolean variable. Boolean variables can be created in several ways. The easiest way is to simply assign True or False to a variable:

set\_to\_true = True

set\_to\_false = False

You can also set a variable equal to a boolean expression.

bool\_one = 5 != 7

bool\_two = 1 + 1 != 2

bool\_three = 3 \* 3 == 9

These variables now contain boolean values, so when you reference them they will only return the True or False values of the expression they were assigned.

>>>bool\_three

True

>>>bool\_four

False

>>>bool\_five

True

Instructions

**1.**

Create a variable named my\_baby\_bool and set it equal to "true".

* my\_baby-bool = ‘true’

**2.**

Check the type of my\_baby\_bool using type(my\_baby\_bool).

You can print the type of a variable by running:

print(type(variable\_name))

* my\_baby-bool = ‘true’
* print(type(my\_baby\_bool))

**3.**

It's not a boolean variable! Boolean values True and False always need to be capitalized and do not have quotation marks.

Create a variable named my\_baby\_bool\_two and set it equal to True.

* my\_baby-bool = ‘true’
* print(type(my\_baby\_bool))

**4.**

Check the type of my\_baby\_bool\_two and make sure you successfully created a boolean variable.

You'll have to print it to get the results to display in the terminal.

You can print the type of a variable by running:

print(type(variable\_name))

* my\_baby-bool = ‘true’
* print(type(my\_baby\_bool))

# If Statements

"Okay okay okay, boolean variables, boolean expressions, blah blah blah, I thought I was learning how to build control flow into my code!"

You are, I promise you!

Understanding boolean variables and expressions is essential because they are the building blocks of conditional statements.

Recall the waking-up example from the beginning of this lesson. The decision-making process of "Is it raining? If so, bring an umbrella" is a conditional statement. Here it is phrased in a different way:

If it is raining then bring an umbrella.

Can you pick out the boolean expression here?

Right, "it is raining" is the boolean expression, and this conditional statement is checking to see if it is True.

If "it is raining" == True then the rest of the conditional statement will be executed and you will bring an umbrella.

This is the form of a conditional statement:

If [it is raining] then [bring an umbrella]

In Python, it looks very similar:

if is\_raining:

bring\_umbrella()

You'll notice that instead of "then" we have a colon, :. That tells the computer that what's coming next is what should be executed if the condition is met. Let's take a look at another conditional statement:

if 2 == 4 - 2:

print("apple")

Will this code print apple to the terminal? Yes, because the condition of the if statement, 2 == 4 - 2 is True.

Let's work through a couple more together:

**Instructions**

**1.**

In the workspace **script.py** there is a function with an if statement. I wrote this function because my coworker Dave kept using my computer without permission and he is a real doofus. It takes user\_name as an input and if the user is Dave it tells him to stay off my computer.

Enter a user name in the field for user\_name and try running the function.

**2.**

Oh no! We got a SyntaxError! This happens when we make a small error in the syntax of the conditional statement.

Read through the error message carefully and see if you can find the error. Then, fix it, and run the code again.

Take a close look at the relational operator in the if statements. Is it correct? Remember, in Python = is used to assign a value to a variable, where == is a relational operator used to see if two items are equal to each other.

**Note:** if you set user\_name to something other than "Dave" you'll notice that None gets printed to the terminal.

This is because the line

print(dave\_check(username))

is printing a function that has not returned a value.

**3.**

Ugh! Dave got around my security and has been logging onto my computer using our coworker Angela's user name, angela\_catlady\_87.

Update the function so it checks for this user name as well and returns

"I know it is you Dave! Go away!"

in response. That'll teach him!

Remember, functions can return all sorts of values. In this case, we want our function to return a string. This should have the form

return "message"

Make sure to copy the string above exactly when writing your function.

**Final Code Solution:**

def dave\_check(user\_name):

if user\_name == "angela\_catlady\_87":

return "I know it is you Dave! Go away!"

# Enter a user name here, make sure to make it a string

user\_name = 'Dave'

print(dave\_check(user\_name))

# Relational Operators II

Now that we've added conditional statements to our toolkit for building control flow, let's explore more ways to create boolean expressions. So far we know two relational operators, equals and not equals, but there are a ton (well, four) more:

* Greater than: >
* Less than: <
* Greater than or equal to: >=
* Less than or equal to: <=

Let's say we're running a movie streaming platform and we want to write a function that checks if our users are over 13 when showing them a PG-13 movie. We could write something like:

def age\_check(age):

if age >= 13:

return True

This function will take the users age and compare it to the number 13. If age is greater than or equal to 13 it will return True.

Let's try some more!

**1.**

Write a function called greater\_than that takes two integer inputs, x and y and returns the value that is greater. If x and y are equal, return the string

"These numbers are the same"

Remember, you define a function using def function\_name():

Also, make sure you copy the string response exactly.

**2.**

The nearby college, Calvin Coolidge's Cool College (or 4C, as the locals call it) requires students to earn 120 credits to graduate. Write a function called graduation\_reqs that takes an input credits and checks if the student has enough credits to graduate. If they do, return the string

"You have enough credits to graduate!"

A basic function with a conditional statement and a return value should look like

def function\_name(parameter):

if [INEQUALITY]:

return some\_value

Community Forums

Get help and ask q

**Final Solution:**

def greater\_than(x, y):

if x > y:

return x

if y > x:

return y

if x == y:

return "These numbers are the same"

def graduation\_reqs(credits):

if credits >= 120:

return "You have enough credits to graduate!"

# Boolean Operators: and

Often, the conditions you want to check in your conditional statement will require more than one boolean expression to cover. In these cases, you can build larger boolean expressions using boolean operators. These operators (also known as logical operators) combine smaller boolean expressions into larger boolean expressions.

There are three boolean operators that we will cover:

* and
* or
* not

Let's start with and.

and combines two boolean expressions and evaluates as True if both its components are True, but False otherwise.

Consider the example

Oranges are a fruit and carrots are a vegetable.

This boolean expression is comprised of two smaller expressions, oranges are a fruit and carrots are a vegetable, both of which are True and connected by the boolean operator and, so the entire expression is True.

Let's look at an example of some AND statements in Python:

>>> (1 + 1 == 2) and (2 + 2 == 4)

True

>>> (1 + 1 == 2) and (2 < 1)

False

>>> (1 > 9) and (5 != 6)

False

>>> (0 == 10) and (1 + 1 == 1)

False

Notice that in the second and third examples, even though part of the expression is True, the entire expression as a whole is False because the other statement is False. The fourth statement is also False because both components are False.

**1.**

Set the variables statement\_one and statement\_two equal to the results of the following boolean expressions:

Statement one:

(2 + 2 + 2 >= 6) and (-1 \* -1 < 0)

Statement two:

(4 \* 2 <= 8) and (7 - 1 == 6)

**2.**

Let's return to Calvin Coolidge's Cool College. 120 credits aren't the only graduation requirement, you also need to have a GPA of 2.0 or higher. Rewrite the graduation\_reqs function so it takes two inputs, gpa and credits, and checks to see if a student meets both requirements using an and statement.

If they do, return the string

"You meet the requirements to graduate!"

Make sure to copy the string exactly!

**Final Solution:**

statement\_one = False

statement\_two = True

def graduation\_reqs(gpa, credits):

if (gpa >= 2.0) and (credits >= 120):

return "You meet the requirements to graduate!"

**NOTE – conditional statements are in ().**

# Boolean Operators: or

The boolean operator or combines two expressions into a larger expression that is True if either component is True.

Consider the statement

Oranges are a fruit or apples are a vegetable.

This statement is composed of two expressions: oranges are a fruit which is True and apples are a vegetable which is False. Because the two expressions are connected by the or operator, the entire statement is True. **Only one component needs to be True for an or statement to be True.**

In English, or implies that if one component is True, then the other component must be False. This is not true in Python. If an or statement has two True components, it is also True.

Let's take a look at a couple example in Python:

>>> True or (3 + 4 == 7)

True

>>> (1 - 1 == 0) or False

True

>>> (2 < 0) or True

True

>>> (3 == 8) or (3 > 4)

False

Notice that each or statement that has at least one True component is True, but the final statement has two False components, so it is False.

**1.**

Set the variables statement\_one and statement\_two equal to the results of the following boolean expressions:

Statement one:

(2 - 1 > 3) or (-5 \* 2 == -10)

Statement two:

(9 + 5 <= 15) or (7 != 4 + 3)

**2.**

The registrars office at Calvin Coolidge's Cool College has another request. They want to send out a mailer with information on the commencement ceremonies to students who have met at least one requirement for graduation (120 credits and 2.0 GPA).

Write a function called graduation\_mailer that takes two inputs, gpa and credits and checks if a student either has 120 or more credits or a GPA 2.0 or higher and if so returns True.

**Final Solution:**

statement\_one = True

statement\_two = True

def graduation\_mailer (gpa, credits):

if (gpa >= 2.0) or (credits >= 120):

return True

**Boolean Operators: not**

The final boolean operator we will cover is not. This operator is straightforward: when applied to any boolean expression it reverses the boolean value. So if we have a True statement and apply a not operator we get a False statement.

not True == False

not False == True

Consider the following statement:

Oranges are not a fruit.

Here, we took the True statement oranges are a fruit and added a not operator to make the False statement oranges are not a fruit.

This example in English is slightly different from the way it would appear in Python because in Python we add the not operator to the very beginning of the statement. Let's take a look at some of those:

>>> not 1 + 1 == 2

False

>>> not 7 < 0

True

**1.**

Set the variables statement\_one and statement\_two equal to the results of the following boolean expressions:

Statement one:

not (4 + 5 <= 9)

Statement two:

not (8 \* 2) != 20 - 4

**2.**

The registrar's office at *Calvin Coolidge's Cool College* has been so impressed with your work so far that they have another task for you. They want you to return to the first function you wrote, graduation\_reqs, and add in several checks using and and not statements.

* If a student meets both requirements the function should return

"You meet the requirements to graduate!"

* If a students GPA is greater or equal to 2.0 but they don't have enough credits the function should return

"You do not have enough credits to graduate."

* If they have enough credits but their GPA is less than 2.0 the function should return

"Your GPA is not high enough to graduate."

* If they do not have enough credits and their GPA is less that 2.0, the function should return

"You do not meet either requirement to graduate!"

Final Solution:

statement\_one = False

statement\_two = True

def graduation\_reqs(gpa, credits):

if (gpa >= 2.0) and (credits >= 120):

return "You meet the requirements to graduate!"

if (gpa >= 2.0) and not (credits >= 120):

return "You do not have enough credits to graduate."

if not (gpa >= 2.0) and (credits >= 120):

return "Your GPA is not high enough to graduate."

if not (gpa >= 2.0) and not (credits >= 120):

return "You do not meet either requirement to graduate!"

Note: The statement “not” goes in front of the condition If / and to ask the question: If not met then…

# Else Statements

As you can tell from your work with Calvin Coolidge's Cool College, once you start including lots of if statements in a function the code becomes a little cluttered and clunky. Luckily, there are other tools we can use to build control flow.

else statements allow us to elegantly describe what we want our code to do when certain conditions are **not** met.

else statements always appear in conjunction with if statements. Consider our waking-up example to see how this works:

if weekday:

wake\_up("6:30")

else:

sleep\_in()

In this way, we can build if statements that execute different code if conditions are or are not met. This prevents us from needing to write if statements for each possible condition, we can instead write a blanket else statement for all the times the condition is not met.

Let's return to our age\_check function for our movie streaming platform. Previously, all it did was check if the user's age was over 13 and if so return True. We can use an else statement to print a message in the event the user is too young to watch the movie.

def age\_check(age):

if age >= 13:

return True

else:

print("Sorry, you must be 13 or older to watch this movie.")

**1.**

Calvin Coolidge's Cool College has **another** request for you. They want you to add an additional check to the graduation\_reqs function. If a student is failing to meet both graduation requirements, they want the function to return:

You do not meet the GPA or the credit requirement for graduation.

Use an else statement to add this to your function.

**Final Solution:**

def graduation\_reqs(gpa, credits):

if (gpa >= 2.0) and (credits >= 120):

return "You meet the requirements to graduate!"

if (gpa >= 2.0) and not (credits >= 120):

return "You do not have enough credits to graduate."

if not (gpa >= 2.0) and (credits >= 120):

return "Your GPA is not high enough to graduate."

else:

return ("You do not meet the GPA or the credit requirement for graduation.")

# Else If Statements

We have if statements, we have else statements, we can also have elif statements.

Now you may be asking yourself, what the heck is an elif statement? It's exactly what it sounds like, "else if". An elif statement checks another condition after the previous if statements conditions aren't met.

We can use elif statements to control the order we want our program to check each of our conditional statements. First, the if statement is checked, then each elif statement is checked from top to bottom, then finally the else code is executed if none of the previous conditions have been met.

Let's take a look at this in practice. The following function will display a "thank you" message after someone donates to a charity: It takes the donation amount and returns a message based on how much was donated.

def thank\_you(donation):

if donation >= 1000:

return "Thank you for your donation! You have achieved platinum donation status!"

elif donation >= 500:

return "Thank you for your donation! You have achieved gold donation status!"

elif donation >= 100:

return "Thank you for your donation! You have achieved silver donation status!"

else:

return "Thank you for your donation! You have achieved bronze donation status!"

Take a second to think about this function. What would happen if all of the elif statements were simply if statements? If you donated $1000.00, then the first three messages would all print because each if condition had been met.

But because we used elif statements, it checks each condition sequentially and only prints one message. If I donate $600.00, the code first checks if that is over $1000.00, which it is not, then it checks if it's over $500.00, which it is, so it prints that message, then because all of the other statements areelif and else, none of them get checked and no more messages get printed.

Try your hand at some other elif statements.

**1.**

Calvin Coolidge's Cool College has noticed that students prefer to get letter grades over GPA numbers. They want you to write a function called grade\_converter that converts an inputted GPA into the appropriate letter grade. Your function should be named grade\_converter, take the input gpa, and convert the following GPAs:

* 4.0 or higher should return "A"
* 3.0 or higher should return "B"
* 2.0 or higher should return "C"
* 1.0 or higher should return "D"
* 0.0 or higher should return "F"

Use elif statements to ensure that only one letter will be printed each time the function is run.

Final Solution:

def grade\_converter (gpa):

if gpa >= 4.0:

return "A"

elif gpa >= 3.0:

return "B"

elif gpa >= 2.0:

return "C"

elif gpa >= 1.0:

return "D"

elif gpa >= 0.0:

return "F"

**TIP: Notice the first statement has an** if **statement and all others have an** elif **statement.**

# Try and Except Statements

if, elif, and else statements aren't the only way to build a control flow into your program. You can use try and except statements to check for possible errors that a user might encounter.

The general syntax of a try and except statement is

try:

# some statement

except ErrorName:

# some statement

First, the statement under try will be executed. If at some point an exception is raised during this execution, such as a NameError or a ValueError and that exception matches the keyword in the except statement, then the try statement will terminate and the except statement will execute.

Let's take a look at this in an application. I want to write a function takes two numbers, a and b as an input and then returns a divided by b. But, there is a possibility that b is zero, which will cause an error, so I want to include a try and except flow to catch this error.

def divides(a,b):

try:

result = a / b

print (result)

except ZeroDivisionError:

print ("Can't divide by zero!")

Now that you see how it works, try to write one yourself.

**1.**

The function in the editor is very simple and serves one purpose: it raises a ValueError.

Try running it by entering raises\_value\_error() into the code editor and hitting run. Remember, unindent this function call so it isn't included in the function itself.

**2.**

Great! Nice error raising! Now let's make that error message a little more palatable.

Write a try statement and an except statement around the line of code that executes the function to catch a ValueError and make the error message print You raised a ValueError!

Your code should follow the structure

try:

# function call

except \_\_\_\_\_\_:

# error message

Where \_\_\_\_\_\_ is the type of error you think will be encountered.

Final Solution:

def raises\_value\_error():

raise ValueError

try:

raises\_value\_error()

except ValueError:

print("You raised a ValueError!")

# Review

Great job! We covered a ton of material in this lesson and you've increased the number of tools in your Python toolkit by several fold. Let's review what you've learned this lesson:

* Boolean expressions are statements that can be either True or False
* A boolean variable is a variable that is set to either True or False.
* You can create boolean expressions using relational operators:
  + Equals: ==
  + Not equals: !=
  + Greater than: >
  + Greater than or equal to: >=
  + Less than: <
  + Less than or equal to: <=
* if statements can be used to create control flow in your code.
* else statements can be used to execute code when the conditions of an if statement are not met.
* elif statements can be used to build additional checks into your if statements
* try and except statements can be used to build error control into your code.

Let's put these skills to the test!

**1.**

The admissions office at Calvin Coolidge's Cool College has heard about your programming prowess and wants to get a piece of it for themselves. They've been inundated with applications and need a way automate the filtering process. They collect three pieces of information for each applicant:

1. Their high school GPA, on a 0.0 - 4.0 scale.
2. Their personal statement, which is given a score on a 1 - 100 scale.
3. The number of extracurricular activities they participate in.

The admissions office has a cutoff point for each category. They want students that have a GPA of 3.0 or higher, a personal statement with a score of 90 or higher, and who participated in 3 or more extracurricular activities.

Write a function called applicant\_selector which takes three inputs, gpa, ps\_score, and ec\_count. If the applicant meets the cutoff point for all three categories, have the function return the string:

"This applicant should be accepted."

**2.**

Great! The admissions office also wants to give students who have a high GPA and a strong personal statement a chance even if they don't participate in enough extracurricular activities.

If an applicant meets the cutoff point for GPA and personal statement score, but not the extracurricular activity count, the function should return the string:

"This applicant should be given an in-person interview."

**3.**

Finally, for all other cases, the function should return the string:

"This applicant should be rejected."

Final Solution

def applicant\_selector (gpa, ps\_score, ec\_count):

if (gpa >= 3.0) and (ps\_score >= 90) and (ec\_count >= 3):

return "This applicant should be accepted."

if (gpa >= 3.0) and (ps\_score >= 90) and not (ec\_count >= 3):

return "This applicant should be given an in-person interview."

else:

return "This applicant should be rejected."