!{sys.executable} -m pip install pyperclip

# For kernel error

python -m ipykernel install --user

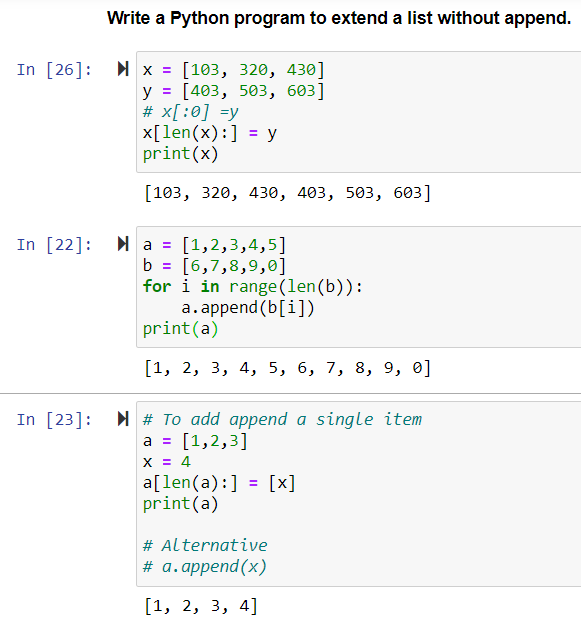
## Lists

Once we have created a list, we may need to add new elements to it, whether it be at the end, beginning, or somewhere in between. Python offers us three different methods to do so. append, extend, and insert

Append

This method adds an element at the end of an existing list.

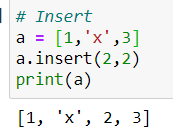
 Here the variable a is our list, and x is the element to add. This expression is equivalent to a[len(a):] = [x].



#### Insert

This method inserts an item at a specified position within the given list.

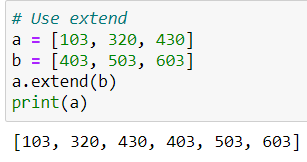
 Here the argument i is the index of the element before which to insert the element x. Thus, a.insert(len(a), x) is the same thing as a.append(x). Although, the power of this method comes in using it to place items somewhere within the list and not at the end. If you only needed to add an element to the end of the list then append works just fine for that, and is faster (which matters for large lists).



#### Extend

This method adds elements (notice its plural!) to a list by adding on all the elements of the iterable you pass to it. The resulting list is one containing all of the elements of both lists.

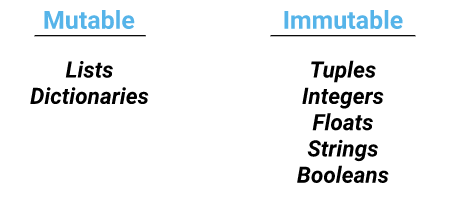
 In this case a is our list and x is an iterable object, such as another list. This method is equivalent to a[len(a):] = x.



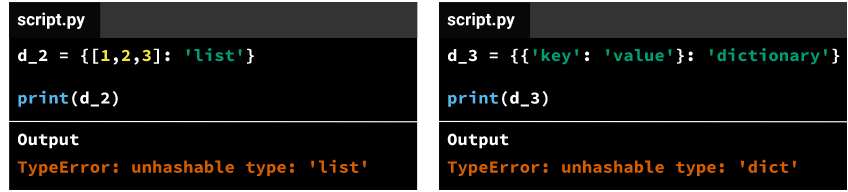
These points show that these three methods are complementary. We must choose which one to use, according to our needs:

* If we want to add an element at the end of a list, we should use append. It is faster and direct.
* If we want to add an element somewhere within a list, we should use insert. It is the only option for this.
* If we want to combine the elements of another iterable to our list, then we should use extend.

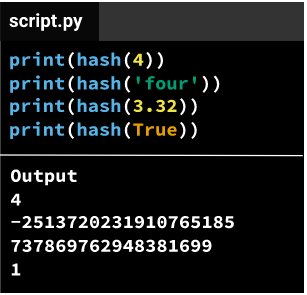
Also append and insert *add only a single element*. Whereas, extend, expands on the initial list by adding the elements of the second list at its end.



except lists and dictionaries. If we use lists or dictionaries as dictionary keys, the computer raises an error:



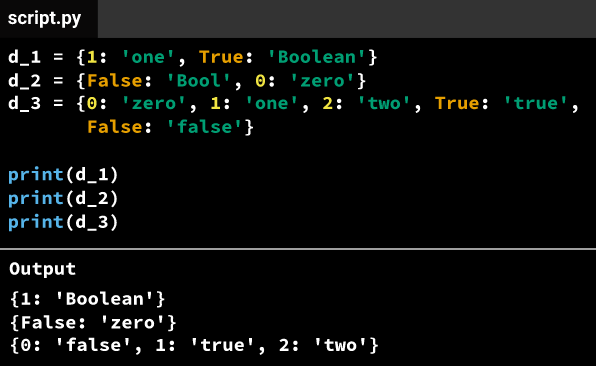
When we populate a dictionary, Python tries to convert each dictionary key to an integer (even if the key is of a data type other than an integer) in the background. Python does the conversion using the hash() command:



the hash() command doesn't transform lists and dictionaries to integers, and returns an error instead.

When we populate a dictionary, we also need to make sure each key in that dictionary is unique. If we use an identical key for two or more different values, Python keeps only the last key-value pair in the dictionary and removes the others — this means that we'll lose data.

An odd "gotcha" is when we mix integers with Booleans as dictionary keys. The hash() command converts the Boolean True to 1, and the Boolean False to 0. This means the Booleans True and False will conflict with the integers 0 and 1. The dictionary keys won't be unique anymore, and Python will only keep the last key-value pair in cases like that.



* **Ctrl + Enter**: run selected cell
* **Shift + Enter**: run cell, select below
* **Alt + Enter**: run cell, insert below

Run the %history -p command to get an understanding of the current state of your program.

[Heading IDs](https://www.markdownguide.org/extended-syntax#heading-ids))

<https://commonmark.org/help/>

<https://www.markdownguide.org/extended-syntax/#tables>

When a function has multiple parameters, there's more than one way to pass in arguments. Consider, for instance, a function named subtract(a, b), which takes a and b as inputs, and returns the result of the subtraction a - b.

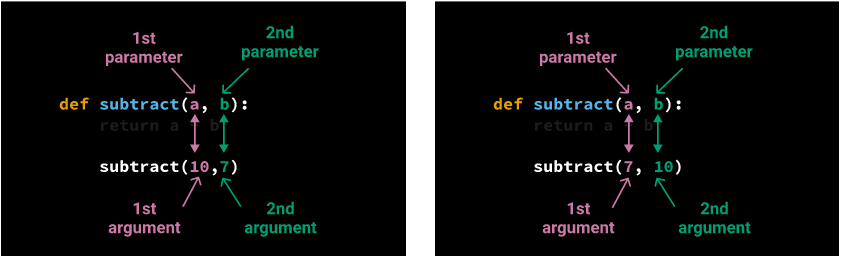
Let's say we want to perform the subtraction 10 - 7. This means that we'll need to pass 10 and 7 in as arguments to the subtract() function. There's more than one way to pass in these two arguments and get the result we want (which is 3 — the result of 10 - 7).

When we use the syntax subtract(a=10, b=7) or subtract(b=7, a=10), we pass in the arguments 10 and 7 using the variable *names* a and b. For this reason, they are called **named arguments**, or more commonly, **keyword arguments**.

When we use keyword arguments, the order we use to pass in the arguments doesn't make any difference. In our example, Python knows exactly that the argument 10 corresponds to the parameter a, and the argument 7 to the parameter b, regardless of whether we use subtract(a=10, b=7) or subtract(b=7, a=10).

This is because when we specify a=10 and b=7, we're crystal clear about what arguments correspond to what parameters, and the order we use to make these specifications doesn't matter anymore.

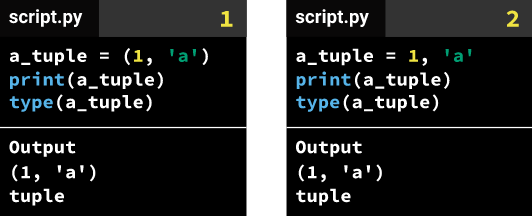
However, when we use subtract(10, 7) or subtract(7, 10), we're not explicit about what arguments correspond to what parameters. To solve this ambiguity, Python maps arguments with parameters *by position*; the first argument will be mapped to the first parameter, and the second argument will be mapped to the second parameter.



Arguments that are passed by position are called **positional arguments**. In the diagram above, we can see the order we use to pass in positional arguments makes a clear difference in terms of argument-parameter mapping, leading to different results in each example (in the example on the left the result will be 3, while on the right the result will be -3).

Positional arguments are often preferred because they involve less typing, and they can speed up our workflow. However, we need to pay extra attention to the order we use to avoid incorrect mappings that can lead to logical errors.

When we create a tuple, surrounding the values with parentheses is optional. It's enough to write the individual values and separate each with a comma. Below, we see two ways of creating a tuple (on the right, we're not using parentheses):



When we use return a\_sum, difference, Python thinks we want the tuple a\_sum, difference returned. This is why multiple variables are returned as tuples.

**String**

The str.title() method returns a copy of the string with the first letter of each word transformed to uppercase (also known as **title case**).

str.replace()

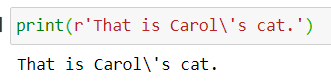
for each\_row in moma:

each\_row[2] = each\_row[2].replace("(","")

each\_row[5] = each\_row[5].replace("(","")

##### **Raw Strings**

A raw string completely ignores all escape characters and prints any backslash that appears in the string.



Raw strings are helpful if you are typing string values that contain many backslashes, such as the strings used for Windows file paths like r'C:\Users\Al\Desktop'

>>> **'Hello' in 'Hello, World'**  
True

>>> **'HELLO' in 'Hello, World'**  
False

**string interpolation**, in which the %s operator inside the string acts as a marker to be replaced by values following the string. One benefit of string interpolation is that str() doesn’t have to be called to convert values to strings.

>>> **name = 'Al'**  
>>> **age = 4000**  
>>> **'My name is %s. I am %s years old.' % (name, age)**  
'My name is Al. I am 4000 years old.'

#### isupper(), and islower()

>>> **spam = 'Hello, world!'**  
>>> **spam.islower()**  
False

>>> **'abc12345'.islower()**  
True  
>>> **'12345'.islower()**  
False  
>>> **'12345'.isupper()**  
False

>>> **'Hello'.upper().lower()**  
'hello'  
>>> **'Hello'.upper().lower().upper()**  
'HELLO'

>>> **'HELLO'.lower().islower()**  
True

**isalpha()** Returns True if the string consists only of letters and isn’t blank

**isalnum()** Returns True if the string consists only of letters and numbers and is not blank

**isdecimal()** Returns True if the string consists only of numeric characters and is not blank

**isspace()** Returns True if the string consists only of spaces, tabs, and newlines and is not blank

**istitle()** Returns True if the string consists only of words that begin with an uppercase letter followed by only lowercase letters

while True:  
    print('Enter your age:')  
    age = input()  
    if age.isdecimal():  
        break  
    print('Please enter a number for your age.')  
  
while True:  
    print('Select a new password (letters and numbers only):')  
    password = input()  
    if password.isalnum():  
        break  
    print('Passwords can only have letters and numbers.')

>>> **'Hello, world!'.startswith('Hello')**  
True  
>>> **'Hello, world!'.endswith('world!')**  
True

#### join() and split() Methods

The join() method is useful when you have a list of strings that need to be joined together into a single string value.

>>> **', '.join(['cats', 'rats', 'bats'])**  
'cats, rats, bats'  
>>> **' '.join(['My', 'name', 'is', 'Simon'])**  
'My name is Simon'  
>>> **'ABC'.join(['My', 'name', 'is', 'Simon'])**  
'MyABCnameABCisABCSimon'

Remember that join() is called on a string value and is passed a list value.

The split() method does the opposite: It’s called on a string value and returns a list of strings.

>>> **'My name is Simon'.split()**  
['My', 'name', 'is', 'Simon']

By default, the string 'My name is Simon' is split wherever whitespace characters such as the space, tab, or newline characters are found. These whitespace characters are not included in the strings in the returned list. You can pass a delimiter string to the split() method to specify a different string to split upon.

#### **Splitting Strings with the partition() Method**

The partition() string method can split a string into the text before and after a separator string.

>>> **'Hello, world!'.partition('w')**  
('Hello, ', 'w', 'orld!')  
>>> **'Hello, world!'.partition('world')**  
('Hello, ', 'world', '!')

If the separator string you pass to partition() occurs multiple times in the string that partition() calls on, the method splits the string only on the first occurrence:

>>> **'Hello, world!'.partition('o')**  
('Hell', 'o', ', world!')

If the separator string can’t be found, the first string returned in the tuple will be the entire string, and the other two strings will be empty:

>>> **'Hello, world!'.partition('XYZ')**  
('Hello, world!', '', '')

#### **Justifying Text with the rjust(), ljust(), and center() Methods**

>>> **'Hello'.rjust(10)**  
'     Hello'

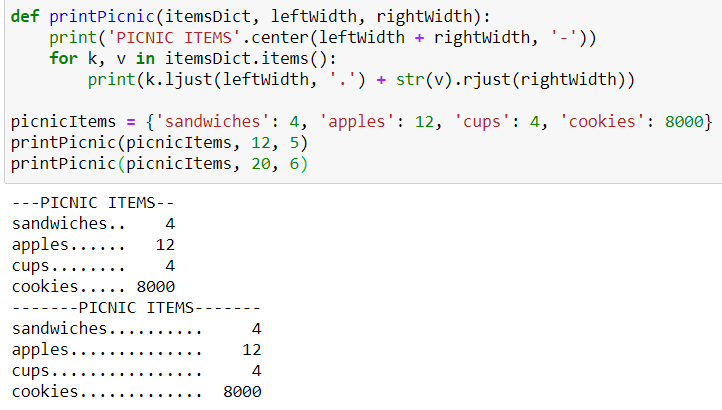
>>> **'Hello'.ljust(10)**  
'Hello     '

'Hello'.rjust(10) says that we want to right-justify 'Hello' in a string of total length 10. 'Hello' is five characters, so five spaces will be added to its left, giving us a string of 10 characters with 'Hello' justified right.

>>> **'Hello'.rjust(20, '\*')**  
'\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Hello'  
>>> **'Hello'.ljust(20, '-')**  
'Hello---------------'

The center() string method works like ljust() and rjust() but centers the text rather than justifying it to the left or right:

>>> **'Hello'.center(20)**  
'       Hello        '  
>>> **'Hello'.center(20, '=')**  
'=======Hello========'



#### **Removing Whitespace with the strip(), rstrip(), and lstrip() Methods**

>>> **spam = '    Hello, World    '**  
>>> **spam.strip()**  
'Hello, World'  
>>> **spam.lstrip()**  
'Hello, World    '  
>>> **spam.rstrip()**  
'    Hello, World'

Optionally, a string argument will specify which characters on the ends should be stripped:

>>> **spam = 'SpamSpamBaconSpamEggsSpamSpam'**  
>>> **spam.strip('ampS')**  
'BaconSpamEggs'

### **Numeric Values of Characters with the ord() and chr() Functions**

Computers store information as bytes—strings of binary numbers, which means we need to be able to convert text to numbers. Because of this, every text character has a corresponding numeric value called a Unicode code point. For example, the numeric code point is 65 for 'A', 52 for '4', and 33 for '!'. You can use the ord() function to get the code point of a one-character string, and the chr() function to get the one-character string of an integer code point.

>>> ord('A')  
65  
>>> ord('4')  
52  
>>> ord('!')  
33  
>>> chr(65)  
'A'

<https://www.youtube.com/watch?v=sgHbC6udIqc&feature=youtu.be>

### **Copying and Pasting Strings with the pyperclip Module**

## **PATTERN MATCHING WITH REGULAR EXPRESSIONS**

Regular expressions use two types of characters:

a) Meta characters: As the name suggests, these characters have a special meaning, similar to \* in wild card.

b) Literals (like a,b,1,2…)

Most letters and characters will simply match themselves. For example, the regular expression test will match the string test exactly. (You can enable a case-insensitive mode that would let this RE match Test or TEST as well).

There are exceptions to this rule; some characters are special metacharacters, and don’t match themselves. Instead, they signal that some out-of-the-ordinary thing should be matched, or they affect other portions of the RE by repeating them or changing their meaning.

Here’s a complete list of the metacharacters;

. ^ $ \* + ? { } [ ] \ | ( )

The first metacharacters we’ll look at are [ and ]. They’re used for specifying a character class, which is a set of characters that you wish to match. Characters can be listed individually, or a range of characters can be indicated by giving two characters and separating them by a '-'. For example, [abc] will match any of the characters a, b, or c; this is the same as [a-c], which uses a range to express the same set of characters. If you wanted to match only lowercase letters, your RE would be [a-z].

Metacharacters are not active inside classes. For example, [akm$] will match any of the characters 'a', 'k', 'm', or '$'; '$' is usually a metacharacter, but inside a character class it’s stripped of its special nature.

You can match the characters not listed within the class by complementing the set. This is indicated by including a '^' as the first character of the class. For example, [^5] will match any character except '5'. If the caret appears elsewhere in a character class, it does not have special meaning. For example: [5^] will match either a '5' or a '^'.

Perhaps the most important metacharacter is the backslash, \. As in Python string literals, the backslash can be followed by various characters to signal various special sequences. It’s also used to escape all the metacharacters so you can still match them in patterns; for example, if you need to match a [ or \, you can precede them with a backslash to remove their special meaning: \[ or \\.

\w matches any alphanumeric character. If the regex pattern is expressed in bytes, this is equivalent to the class [a-zA-Z0-9\_].

\d

Matches any decimal digit; this is equivalent to the class [0-9].

\D

Matches any non-digit character; this is equivalent to the class [^0-9].

\s

Matches any whitespace character; this is equivalent to the class [ \t\n\r\f\v].

\S

Matches any non-whitespace character; this is equivalent to the class [^ \t\n\r\f\v].

\w

Matches any alphanumeric character; this is equivalent to the class [a-zA-Z0-9\_].

\W

Matches any non-alphanumeric character; this is equivalent to the class [^a-zA-Z0-9\_].

These sequences can be included inside a character class. For example, [\s,.] is a character class that will match any whitespace character, or ',' or '.'.

The final metacharacter in this section is .. It matches anything except a newline character, and there’s an alternate mode ([re.DOTALL](https://docs.python.org/3/library/re.html" \l "re.DOTALL" \o "re.DOTALL)) where it will match even a newline. . is often used where you want to match “any character”.

The most common uses of regular expressions are:

* Search a string (search and match)
* Finding a string (findall)
* Break string into a sub strings (split)
* Replace part of a string (sub)

The ‘re’ package provides multiple methods to perform queries on an input string. Here are the most commonly used methods:

1. re.match()
2. re.search()
3. re.findall()
4. re.split()
5. re.sub()
6. re.compile()

### re.match(pattern, string):

#### **Creating Regex Objects**

All the regex functions in Python are in the re module.

Passing a string value representing your regular expression to re.compile() returns a Regex **pattern object** (or simply, a Regex object).

>>> **phoneNumRegex = re.compile(r'\d\d\d-\d\d\d-\d\d\d\d')**

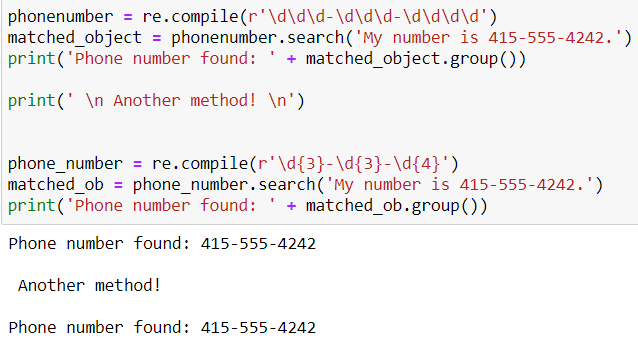
\d in a regex stands for a digit character—that is, any single numeral from 0 to 9.

Remember to use a **raw** string.

Now the phoneNumRegex variable contains a Regex object.

#### **Matching Regex Objects**

A Regex object’s search() method searches the string it is passed for any matches to the regex. The search() method will return None if the regex pattern is not found in the string. If the pattern is found, the search() method returns a Match object, which have a group() method that will return the actual matched text from the searched string.



Here, we pass our desired pattern to re.compile() and store the resulting Regex object in phonenum. Then we call search() on phonenum and pass search() the string we want to match for during the search. The result of the search gets stored in the variable matched\_object. In this example, we know that our pattern will be found in the string, so we know that a Match object will be returned. Knowing that matched\_object contains a Match object and not the null value None, we can call group() on matched\_object to return the match. Writing matched\_object. group() inside our print() function call displays the whole match, 415-555-4242.

**Map, Filter, Reduce**

Map, Filter, and Reduce are paradigms of functional programming. They allow the programmer (you) to write simpler, shorter code, without necessarily needing to bother about intricacies like loops and branching.

Essentially, these three functions allow you to apply a function across a number of iterables, in one full swoop. map and filter come built-in with Python (in the \_\_builtins\_\_ module) and require no importing. reduce, however, needs to be imported as it resides in the functools module.

#### **Map**

**map() passes each element in the iterable through a function and returns the result of all elements having passed through the function**

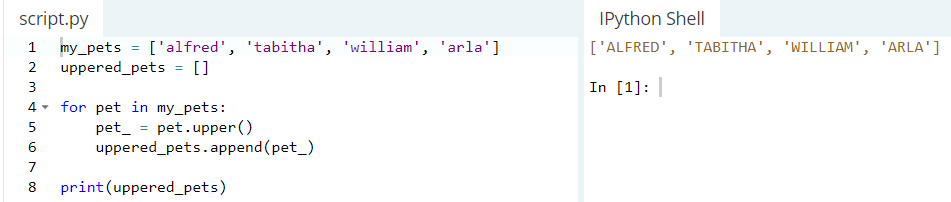
The map() function in python has the following syntax:

map(func, \*iterables)

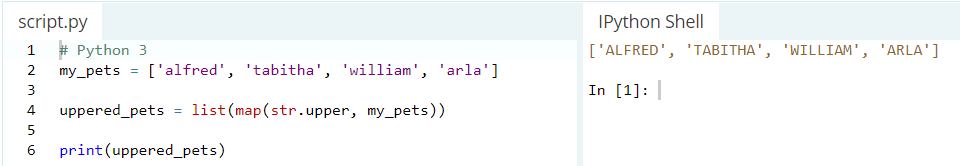
Where func is the function on which each element in iterables (as many as they are) would be applied on. Notice the asterisk(\*) on iterables? It means there can be as many iterables as possible, in so far func has that exact number as required input arguments.

 In Python 3, the function returns a map object which is a generator object. To get the result as a list, the built-in list() function can be called on the map object. i.e. list(map(func, \*iterables))

The number of arguments to func must be the number of iterables listed.



With map() functions,

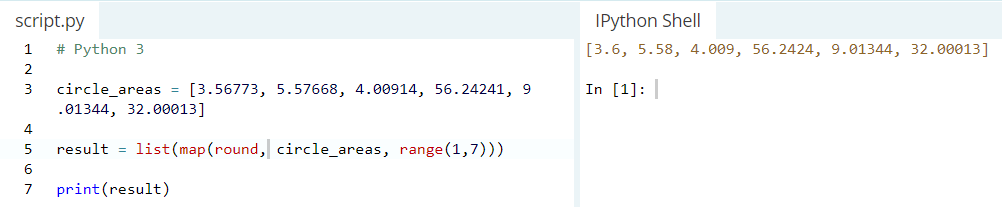


Which would also output the same result. Note that using the defined map() syntax above, func in this case is str.upper and iterables is the my\_pets list -- just one iterable. Also note that we did not call the str.upper function (doing this: str.upper()), as the map function does that for us on each element in the *my\_pets* list.

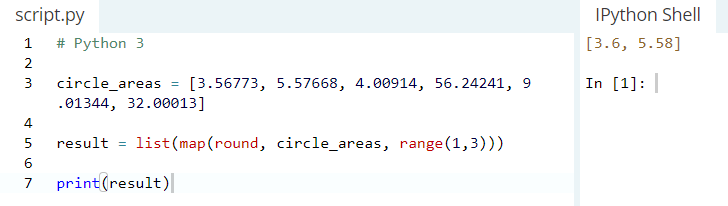
What's more important to note is that the str.upper function requires only **one** argument by definition and so we passed just **one** iterable to it. So, if the function you're passing requires two, or three, or n arguments, then you need to pass in two, three or n iterables to it.

Say I have a list of circle areas that I calculated somewhere, all in five decimal places. And I need to round each element in the list up to its position decimal places, meaning that I have to round up the first element in the list to one decimal place, the second element in the list to two decimal places, the third element in the list to three decimal places, etc.

Python has round() built-in function that takes two arguments -- the number to round up and the number of decimal places to round the number up to. So, since the function requires **two** arguments, we need to pass in **two** iterables.

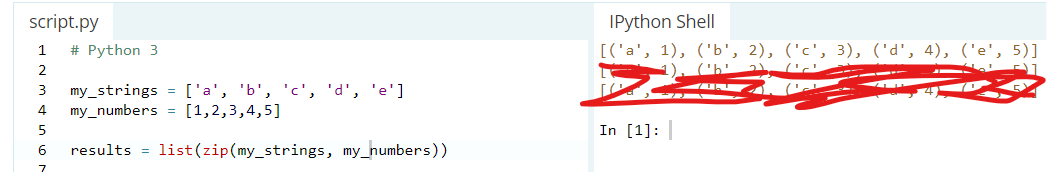


if you evaluate result = list(map(round, circle\_areas, range(1,3)))

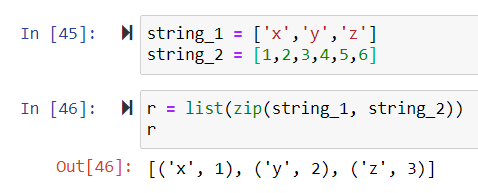


**zip() function**

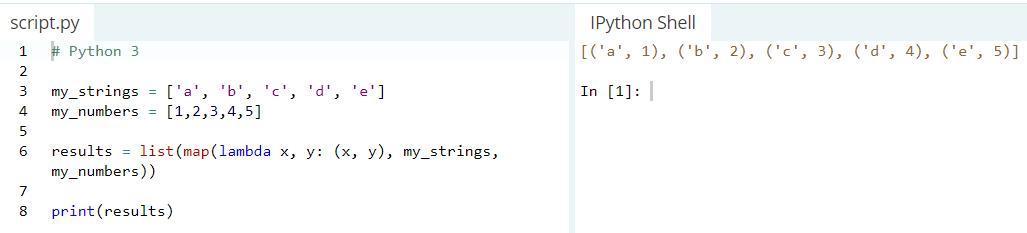
The zip() function is a function that takes a number of iterables and then creates a tuple containing each of the elements in the iterables. Like map(), in Python 3, it returns a generator object, which can be easily converted to a list by calling the built-in list function on it.



if string\_1 and string\_2 are not of the same length?



**custom zip() function!**



**Filter**

filter(), requires the function to return boolean values (true or false) and then passes each element in the iterable through the function, "filtering" away those that are false.

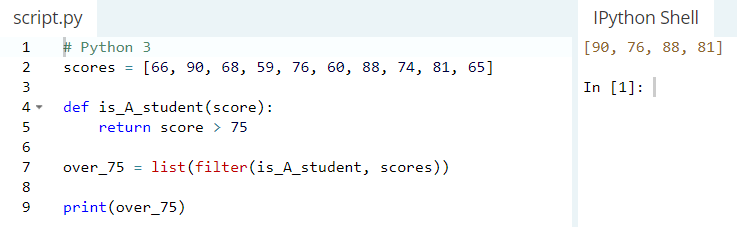
It has the following syntax:

filter(func, iterable)

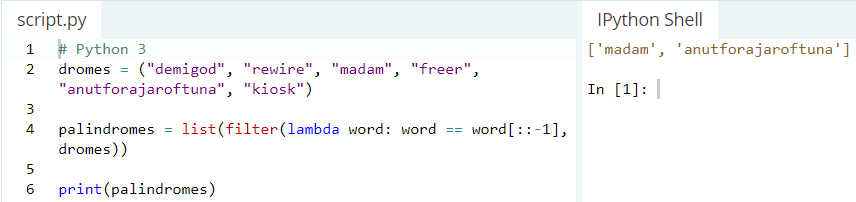
The following points are to be noted regarding filter():

1. Unlike map(), only one iterable is required.
2. The func argument is required to return a boolean type. If it doesn't, filter simply returns the iterable passed to it. Also, as only one iterable is required, it's implicit that func must only take one argument.
3. filter passes each element in the iterable through func and returns **only** the ones that evaluate to true. I mean, it's right there in the name -- a "filter".

The following is a list (iterable) of the scores of 10 students in a Chemistry exam. Let's filter out those who passed with scores more than 75...using filter.



The next example will be a palindrome detector. A "palindrome" is a word, phrase, or sequence that reads the same backwards as forwards. Let's filter out words that are palindromes from a tuple (iterable) of suspected palindromes.



#### **Reduce**

reduce applies a function **of two arguments** cumulatively to the elements of an iterable, optionally starting with an initial argument. It has the following syntax:

reduce(func, iterable[, initial])

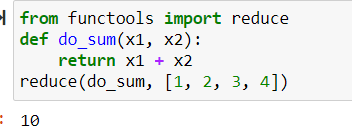
Where func is the function on which each element in the iterable gets cumulatively applied to, and initial is the optional value that gets placed before the elements of the iterable in the calculation, and serves as a default when the iterable is empty.

following should be noted about reduce():

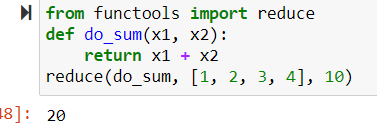
1. func requires two arguments, the first of which is the first element in iterable (if initial is not supplied) and the second the second element in iterable. If initial is supplied, then it becomes the first argument to func and the first element in iterable becomes the second element.

2. reduce "reduces" iterable into a single value.

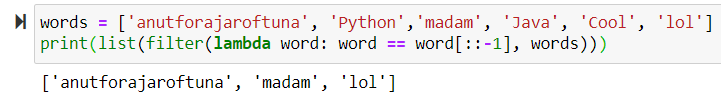
Without initial



With initial



**Lambda Expression**



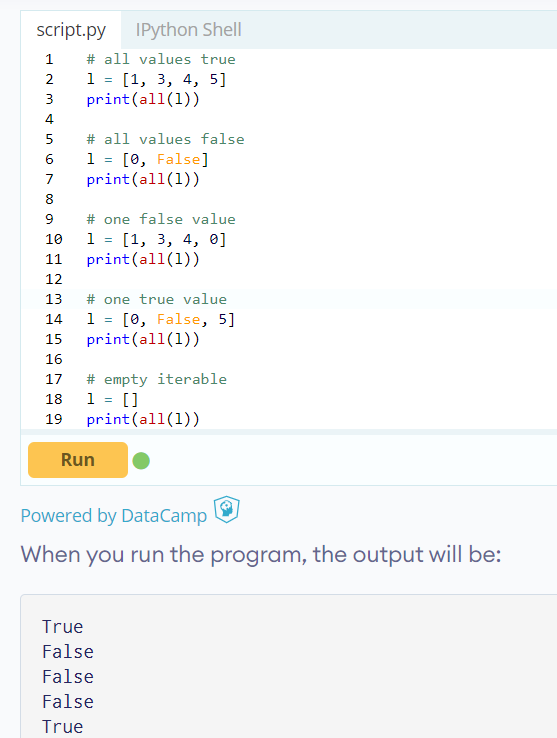
## **all()**

The all() method takes a single parameter:

* **iterable** - any iterable ([list](https://www.programiz.com/python-programming/list), [tuple](https://www.programiz.com/python-programming/tuple), [dictionary](https://www.programiz.com/python-programming/dictionary), etc.) which contains the elements

The all() method returns:

* **True** - If all elements in an iterable are true
* **False** - If any element in an iterable is false

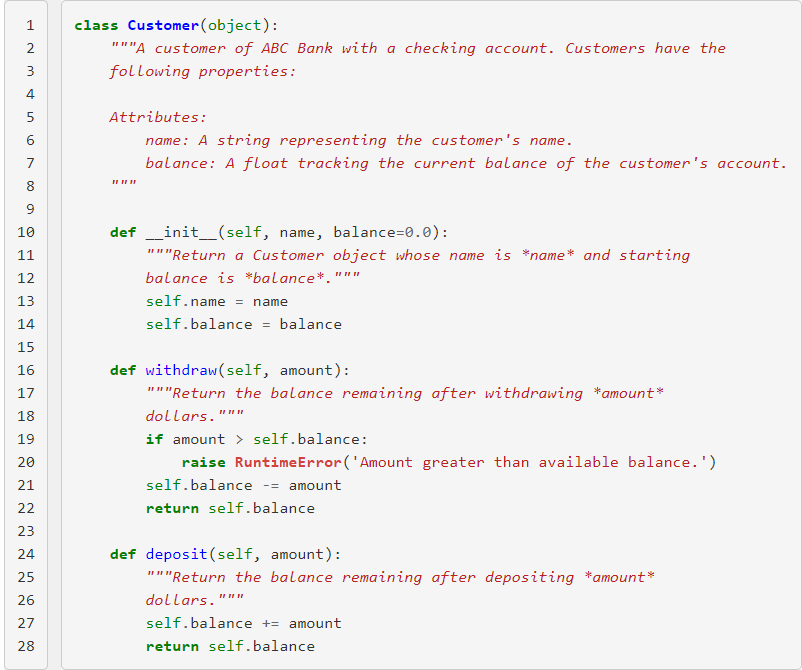


**OOP:**

Classes and objects are the two main aspects of object oriented programming. A **class** creates a new type where **objects** are **instances** of the class. An analogy is that you can have variables of type int which translates to saying that variables that store integers are variables which are instances (objects) of the int class.

And what is a class? Simply a logical grouping of data and functions (the latter of which are frequently referred to as "methods" when defined within a class).

Classes can be thought of as blueprints for creating objects. When I define a Customer class using the class keyword, I haven't actually created a customer. Instead, what I've created is a sort of instruction manual for constructing "customer" objects.



To create a Customer object, we call the class's \_\_init\_\_ method with the proper number of arguments.

So, to use the "blueprint" that we created by defining the class Customer (which is used to create Customer objects), we call the class name almost as if it were a function: jeff = Customer('Jeff Knupp', 1000.0). This line simply says "use the Customer blueprint to create me a new object, which I'll refer to as jeff."

The jeff object, known as an instance, is the realized version of the Customer class. Before we called Customer(), no Customer object existed. We can, of course, create as many Customer objects as we'd like. There is still, however, only one Customer class, regardless of how many instances of the class we create.

Objects can store data using ordinary variables that *belong* to the object. Variables that belong to an object or class are referred to as **fields**. Objects can also have functionality by using functions that *belong* to a class. Such functions are called **methods** of the class. This terminology is important because it helps us to differentiate between functions and variables which are independent and those which belong to a class or object. Collectively, the fields and methods can be referred to as the **attributes** of that class.

Fields are of two types - they can belong to each instance/object of the class or they can belong to the class itself. They are called **instance variables** and **class variables** respectively.

## The self

Class methods have only one specific difference from ordinary functions - they must have an extra first name that has to be added to the beginning of the parameter list, but you **do not** give a value for this parameter when you call the method, Python will provide it. This particular variable of name self refers to the object itself.

So, when we say def withdraw(self, amount):, we're saying, "here's how you withdraw money from a Customer object (which we'll call self) and a dollar figure (which we'll call amount). self is the instance of the Customer that withdraw is being called on.

So, jeff.withdraw(100.0) is just shorthand for Customer.withdraw(jeff, 100.0)

You must be wondering how Python gives the value for self and why you don't need to give a value for it. An example will make this clear. Say you have a class called MyClass and an instance of this class called myobject. When you call a method of this object as myobject.method(arg1, arg2), this is automatically converted by Python into MyClass.method(myobject, arg1, arg2) - this is all the special self is about.

## The \_\_init\_\_ method

The \_\_init\_\_ method is run as soon as an object of a class is instantiated (i.e. created). The method is useful to do any initialization (i.e. passing initial values to your object) you want to do with your object.



This may look like a reasonable alternative; we simply need to call set\_balance before we begin using the instance. There's no way, however, to communicate this to the caller. Even if we document it extensively, we can't force the caller to call jeff.set\_balance(1000.0) before calling jeff.withdraw(100.0). Since the jeff instance doesn't even have a balance attribute until jeff.set\_balance is called, this means that the object hasn't been "fully" initialized.

Don't introduce a new attribute outside of the \_\_init\_\_ method, otherwise you've given the caller an object that isn't fully initialized.

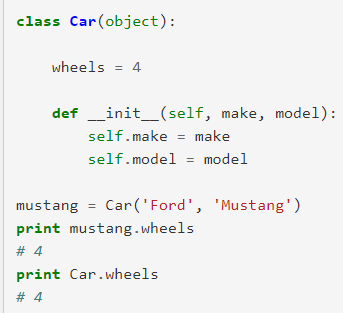
## **Instance Attributes and Methods**

A function defined in a class is called a "method". Methods have access to all the data contained on the instance of the object; they can access and modify anything previously set on self. Because they use self, they require an instance of the class in order to be used. For this reason, they're often referred to as "instance methods".

If there are "instance methods", then surely there are other types of methods as well, right?

### **Static Methods**

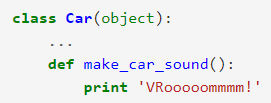
Class attributes are attributes that are set at the class-level, as opposed to the instance-level. Normal attributes are introduced in the \_\_init\_\_ method, but some attributes of a class hold for all instances in all cases. For example, consider the following definition of a Car object:



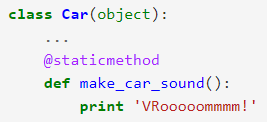
A Car always has four wheels, regardless of the make or model. Instance methods can access these attributes in the same way they access regular attributes: through self (i.e. self.wheels).

There is a class of methods, though, called *static methods*, that don't have access to self. Just like class attributes, they are methods that work without requiring an instance to be present. Since instances are always referenced through self, static methods have no self parameter.

The following would be a valid static method on the Car class:

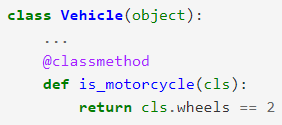


To make it clear that this method should not receive the instance as the first parameter (i.e. self on "normal" methods), the @staticmethod decorator is used, turning our definition into:



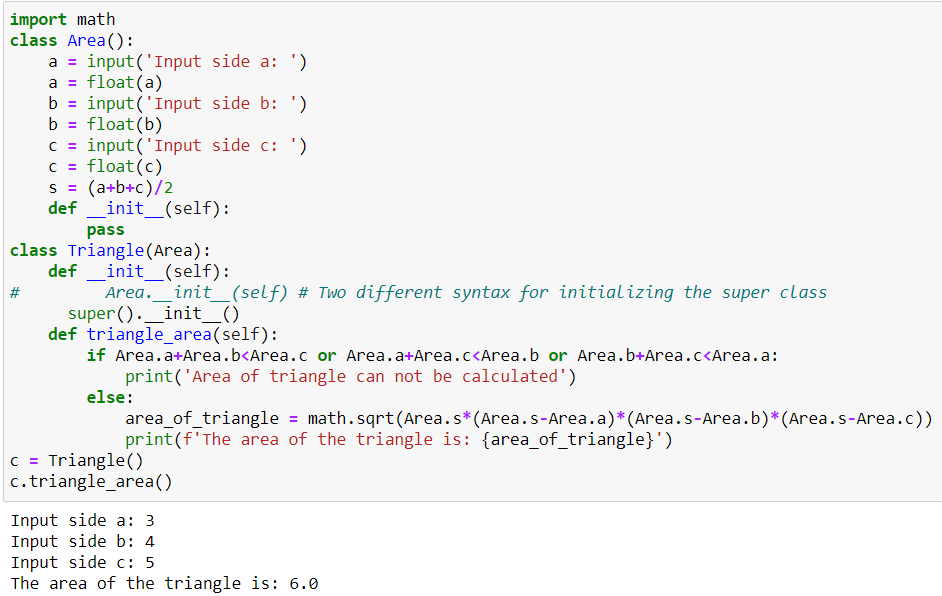
### **Class Methods**

A variant of the static method is the class method. Instead of receiving the instance as the first parameter, it is passed the class. It, too, is defined using a decorator:



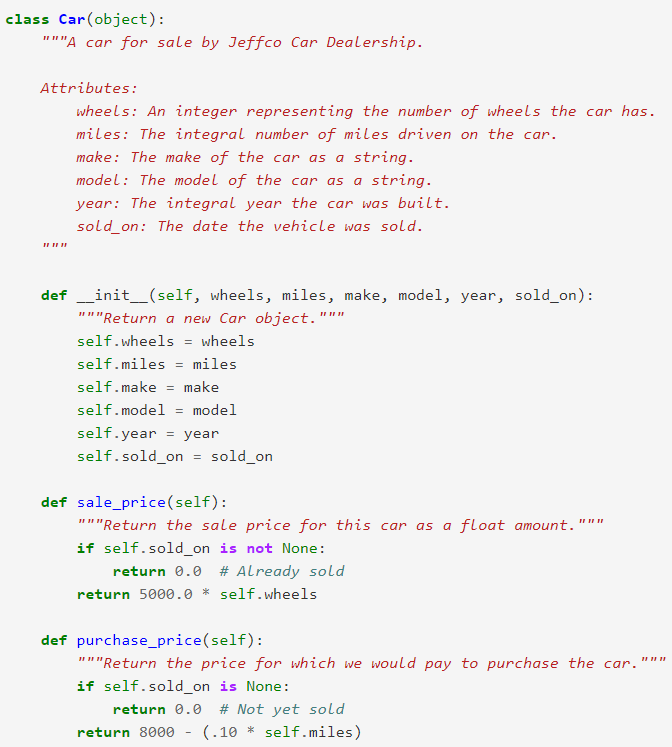
Class methods may not make much sense right now, but that's because they're used most often in connection with our next topic: inheritance.

\_\_inheritance\_\_

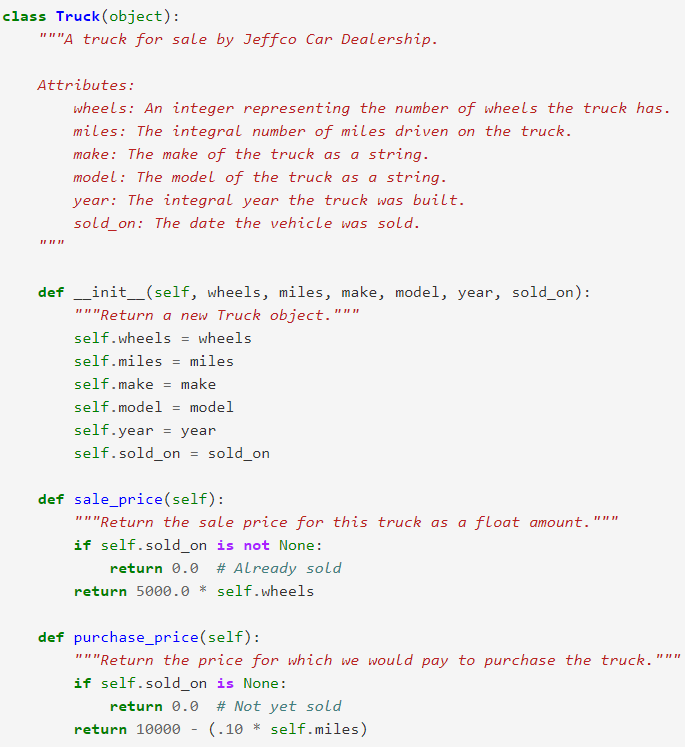


**Inheritance** is the process by which a "child" class derives the data and behaviour of a "parent" class.

Imagine we run a car dealership. We sell all types of vehicles, from motorcycles to trucks. We set ourselves apart from the competition by our prices. Specifically, how we determine the price of a vehicle on our lot: $5,000 x number of wheels a vehicle has. We love buying back our vehicles as well. We offer a flat rate - 10% of the miles driven on the vehicle. For trucks, that rate is $10,000. For cars, $8,000. For motorcycles, $4,000.



Now that we've got the Car class, perhaps we should create a Truck class? Let's follow the same pattern we did for car:

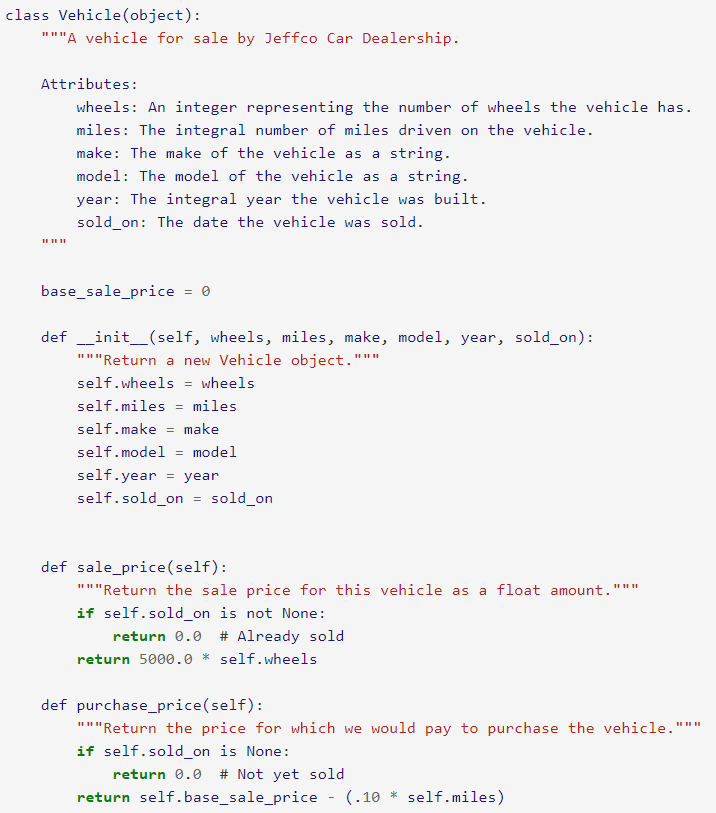


That's almost identical to the car class. One of the most important rules of programming (in general, not just when dealing with objects) is "DRY" or "**D**on't **R**epeat **Y**ourself. We've definitely repeated ourselves here. In fact, the Car and Truck classes differ only by a single character (aside from comments).

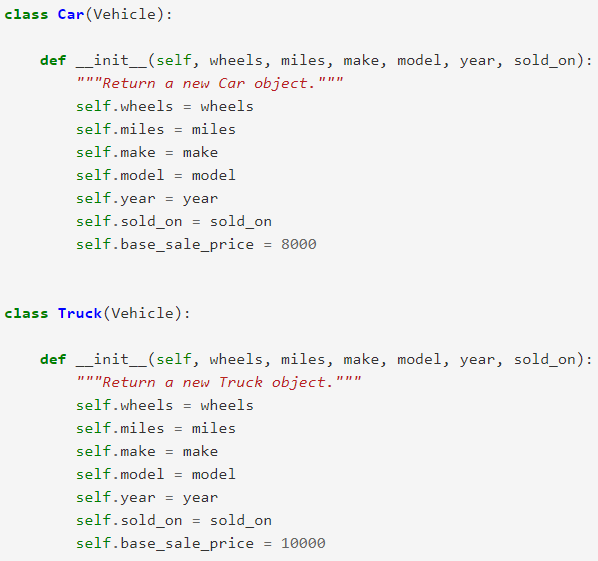
Our main problem is that we raced straight to the concrete: Cars and Trucks are real things, tangible objects that make intuitive sense as classes. However, they share so much data and functionality in common that it seems there must be an abstraction we can introduce here. Indeed there is: the notion of Vehicles.

### **Abstract Classes**

A Vehicle is not a real-world object. Rather, it is a concept that some real-world objects (like cars, trucks, and motorcycles) embody. We would like to use the fact that each of these objects can be considered a vehicle to remove repeated code. We can do that by creating a Vehicle class:



Now we can make the Car and Truck class inherit from the Vehicle class



This works, but has a few problems. First, we're still repeating a lot of code. We'd ultimately like to get rid of **all** repetition. Second, and more problematically, we've introduced the Vehicle class, but should we really allow people to create Vehicle objects (as opposed to Cars or Trucks)? A Vehicle is just a concept, not a real thing, so what does it mean to say the following:



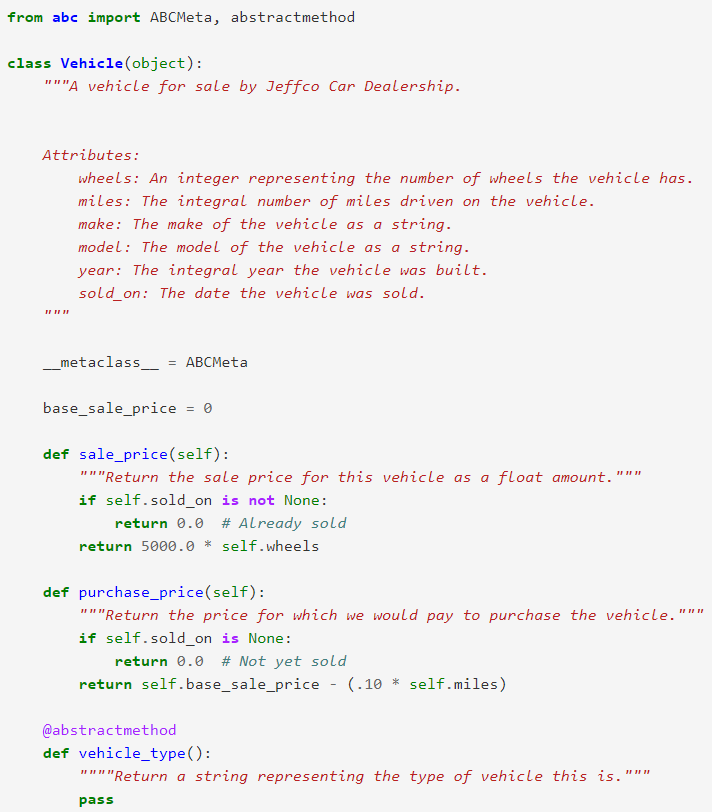
A Vehicle doesn't have a base\_sale\_price, only the individual child classes like Car and Truck do. The issue is that Vehicle should really be an Abstract Base Class. Abstract Base Classes are classes that are only meant to be inherited from; you can't create instance of an ABC. That means that, if Vehicle is an ABC, the following is illegal:



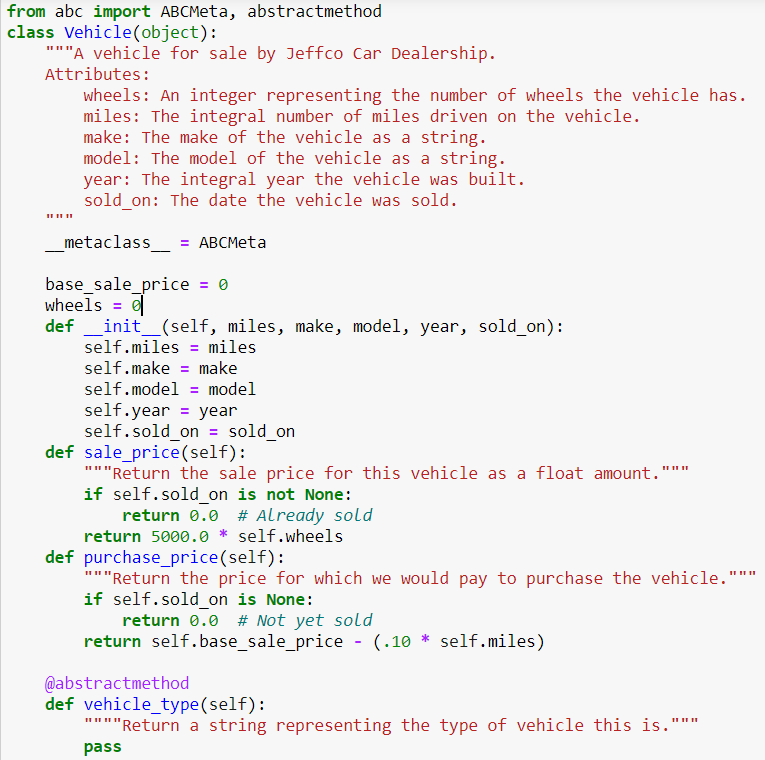
It makes sense to disallow this, as we never meant for vehicles to be used directly. We just wanted to use it to abstract away some common data and behavior. So how do we make a class an ABC?

Simple! The abc module contains a metaclass called ABCMeta. Setting a class's metaclass to ABCMeta and making one of its methods virtual makes it an ABC.

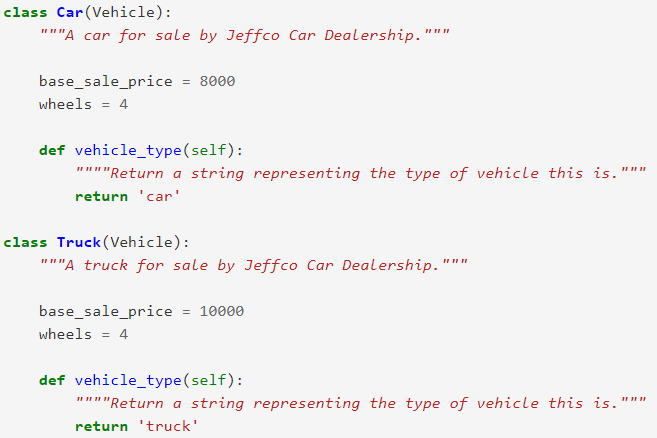
A virtual method is one that the ABC says must exist in child classes, but doesn't necessarily actually implement. For example, the Vehicle class may be defined as follows:



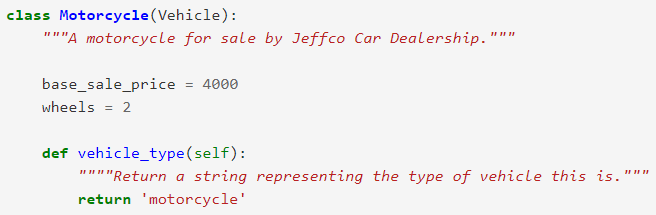
Now, since vehicle\_type is an abstractmethod, we can't directly create an instance of Vehicle. As long as Car and Truck inherit from Vehicle **and** define vehicle\_type, we can instantiate those classes just fine.



Now the Car and Truck classes become:

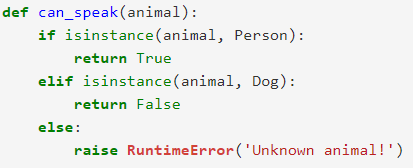


This fits perfectly with our intuition: as far as our system is concerned, the only difference between a car and truck is the base sale price. Defining a Motorcycle class, then, is similarly simple:



Even though it seems like we used inheritance to get rid of duplication, what we were *really* doing was simply providing the proper level of abstraction. And *abstraction* is the key to understanding inheritance. We've seen how one side-effect of using inheritance is that we reduce duplicated code, but what about from the *caller's perspective*. How does using inheritance change that code?

Quite a bit, it turns out. Imagine we have two classes, Dog and Person, and we want to write a function that takes either type of object and prints out whether or not the instance in question can speak (a dog can't, a person can). We might write code like the following:



That works when we only have two types of animals, but what if we have twenty, or *two hundred*? That if...elif chain is going to get quite long.

The key insight here is that can\_speak shouldn't care what type of animal it's dealing with, the animal class itself should tell *us* if it can speak. By introducing a common base class, Animal, that defines can\_speak, we relieve the function of it's type-checking burden. Now, as long as it knows it was an Animal that was passed in, determining if it can speak is trivial:



This works because Person and Dog (and whatever other classes we create to derive from Animal) follow the Liskov Substitution Principle. This states that we should be able to use a child class (like Person or Dog) wherever a parent class (Animal) is expected an everything will work fine. This sounds simple, but it is the basis for a powerful concept we'll discuss in a future article: interfaces.

Python has two built-in functions that work with inheritance:

* Use [isinstance()](https://docs.python.org/3/library/functions.html" \l "isinstance" \o "isinstance) to check an instance’s type: isinstance(obj, int) will be True only if obj.\_\_class\_\_ is [int](https://docs.python.org/3/library/functions.html#int) or some class derived from [int](https://docs.python.org/3/library/functions.html#int).
* Use [issubclass()](https://docs.python.org/3/library/functions.html#issubclass) to check class inheritance: issubclass(bool, int) is True since [bool](https://docs.python.org/3/library/functions.html#bool) is a subclass of [int](https://docs.python.org/3/library/functions.html#int). However, issubclass(float, int) is False since [float](https://docs.python.org/3/library/functions.html#float) is not a subclass of [int](https://docs.python.org/3/library/functions.html#int).

## The errors and exception handling methods

Types of errors:

* 1. Compile time error

Ex: Syntax errors

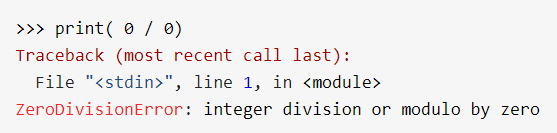
* 1. Logical error
  2. Run time error

Ex: Divide by zero

In Python, an error can be a syntax error or an exception.

Syntax errors also known as parsing errors.

Below one is an exception error.

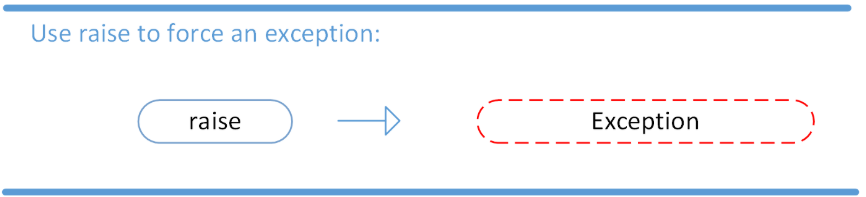


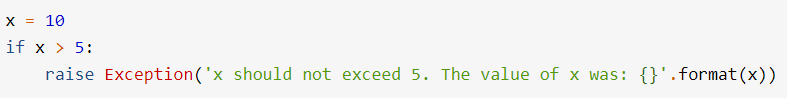
The last line of the message indicated what type of exception error you ran into.

Instead of showing the message exception error, Python details what type of exception error was encountered. In this case, it was a ZeroDivisionError. Python comes with [various built-in exceptions](https://docs.python.org/3/library/exceptions.html) as well as the possibility to create self-defined exceptions.

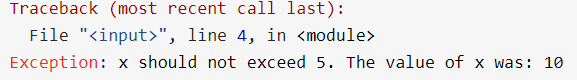
## Raising an Exception

We can use raise to throw an exception if a condition occurs. The statement can be complemented with a custom exception.





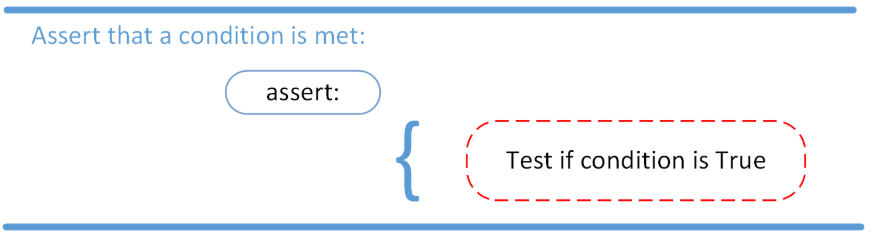
When you run this code, the output will be the following:



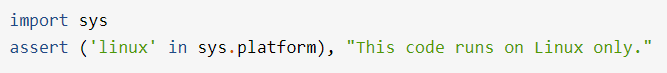
## The AssertionError Exception

Instead of waiting for a program to crash midway, we assert that a certain condition is met.

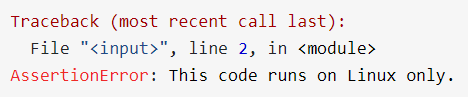
If this condition turns out to be True, then that is excellent! The program can continue. If the condition turns out to be False, you can have the program throw an AssertionError exception.



Have a look at the following example, where it is asserted that the code will be executed on a Linux system:



If you run this code on a Linux machine, the assertion passes. If you were to run this code on a Windows machine, the outcome of the assertion would be False and the result would be the following:

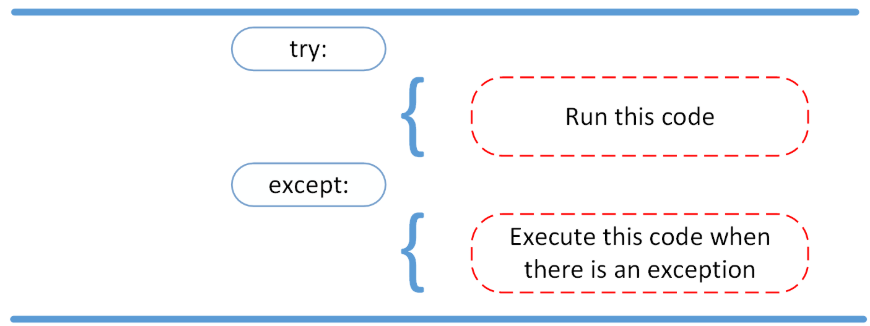


In this example, throwing an AssertionError exception is the last thing that the program will do. The program will come to halt and will not continue.

What if that is not what you want?

## The try and except Block: Handling Exceptions

The try and except block in Python is used to catch and handle exceptions. Python executes code following the try statement as a “normal” part of the program. The code that follows the except statement is the program’s response to any exceptions in the preceding try clause.

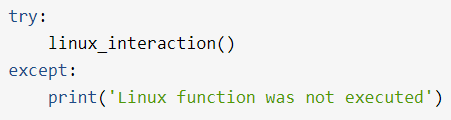


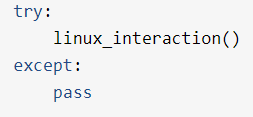
when syntactically correct code runs into an error, Python will throw an exception error. This exception error will crash the program if it is unhandled.

The except clause determines how your program responds to exceptions.



The linux\_interaction() can only run on a Linux system. The assert in this function will throw an AssertionError exception if you call it on an operating system other then Linux.

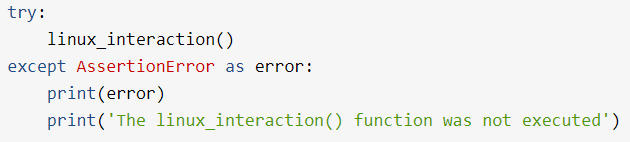
****You can give the function a try using the following code:

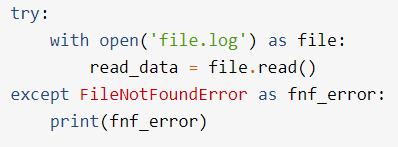
 better alternative🡪

When an exception occurs in a program running this function, the program will continue as well as inform you about the fact that the function call was not successful.

What you did not get to see was the type of error that was thrown as a result of the function call. In order to see exactly what went wrong, you would need to catch the error that the function threw.

The following code is an example where you capture the AssertionError and output that message to screen:

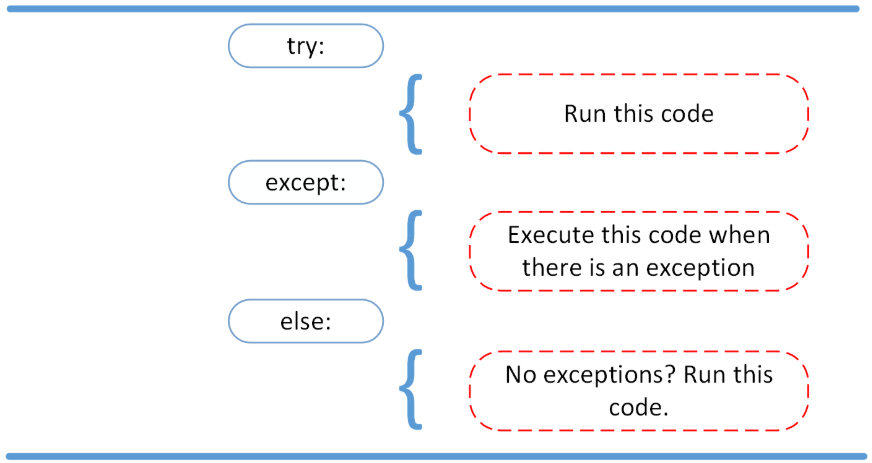




code in the try clause will stop as soon as an exception is encountered.

* A try clause is executed up until the point where the first exception is encountered.
* Inside the except clause, or the exception handler, you determine how the program responds to the exception.
* You can anticipate multiple exceptions and differentiate how the program should respond to them.
* [Avoid using bare except clauses.](https://realpython.com/the-most-diabolical-python-antipattern/)

## The else Clause

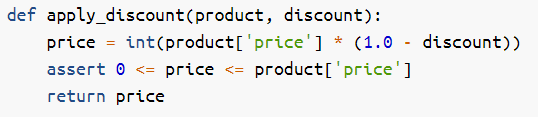
In Python, using the else statement, you can instruct a program to execute a certain block of code only in the absence of exceptions.

When there is an exception, else will not be executed. Only in case of successful try block execution, else will get executed.

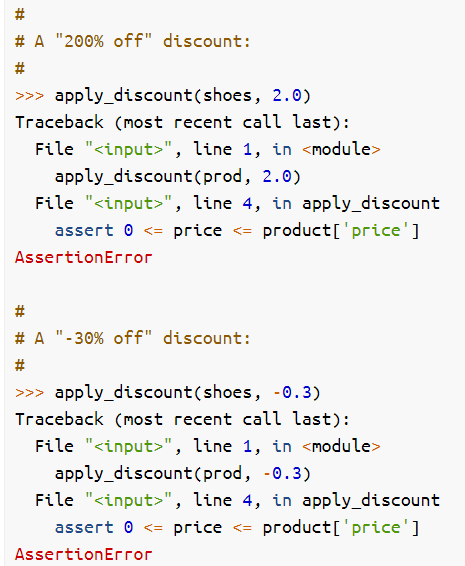
## Cleaning Up After Using finally

Imagine that you always had to implement some sort of action to clean up after executing your code. Python enables you to do so using the finally clause.

## Assert in Python — An Example



Notice the assert statement in there? It will guarantee that, no matter what, discounted prices cannot be lower than $0 and they cannot be higher than the original price of the product.

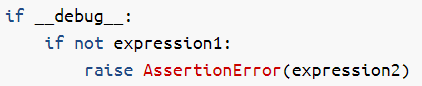


## Assert Syntax

"assert" expression1 ["," expression2]

In this case expression1 is the condition we test, and the optional expression2 is an error message that’s displayed if the assertion fails.

At execution time, the Python interpreter transforms each assert statement into roughly the following:



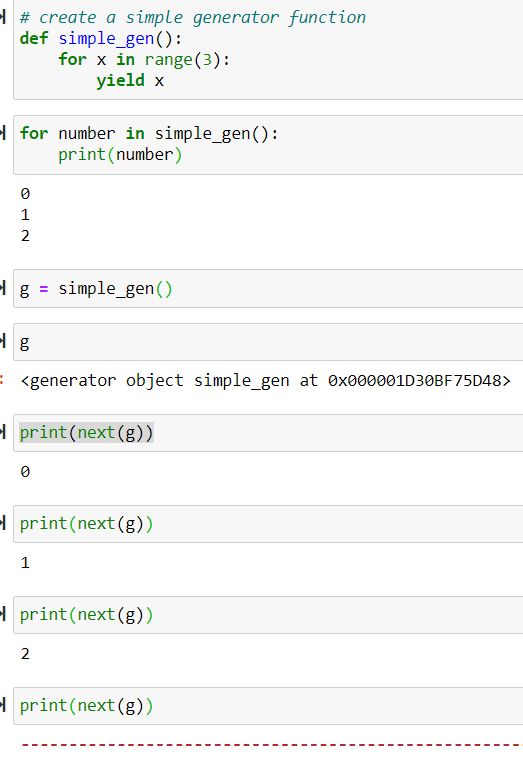
Decorators:

Generators:

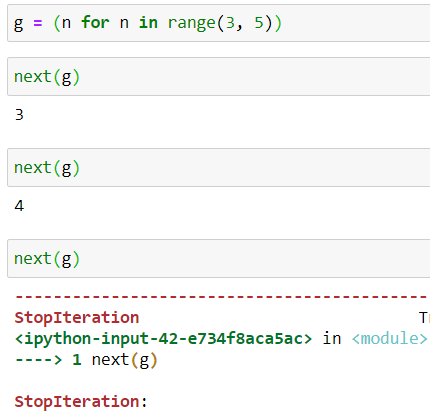
A generator is simply a function which returns an object on which you can call next, such that for every call it returns some value, until it raises a StopIteration exception, signaling that all values have been generated. Such an object is called an iterator.

In most aspects, a generator function will appear very similar to a normal function. The main difference is when a generator function is compiled they become an object that supports an iteration protocol. That means when they are called in your code they don't actually return a value and then exit. Instead, generator functions will automatically suspend and resume their execution and state around the last point of value generation. The main advantage here is that instead of having to compute an entire series of values up front, the generator computes one value and then suspends its activity awaiting the next instruction. This feature is known as state suspension.

Normal functions return a single value using return, just like in Java. In Python, however, there is an alternative, called yield. Using yield anywhere in a function makes it a generator.



Likewise, there are generator\_expressions which provide a means to succinctly describe certain common types of generators:



**Generator comprehension:**

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

gencomp = (item **for** item **in** my\_list **if** item > 3)

**for** item **in** gencomp:

print(item)



A generator object is generated once, but its code is not run all at once. Only calls to next actually execute (part of) the code. Execution of the code in a generator stops once a yield statement has been reached, upon which it returns a value. The next call to next then causes execution to continue in the state in which the generator was left after the last yield. This is a fundamental difference with regular functions: those always start execution at the "top" and discard their state upon returning a value.

Generators are iterators, a kind of iterable **you can only iterate over once**. Generators do not store all the values in memory, **they generate the values on the fly.**