**CHAPTER 8**

**PYTHON ADVANCED TOPICS**

**Python Iterators**

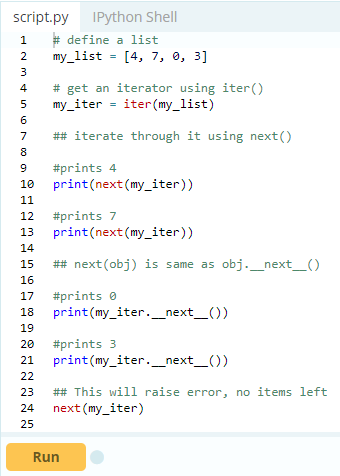
Iterators are objects that can be iterated upon. In this tutorial, you will learn how iterator works and how you can build your own iterator using \_\_iter\_\_ and \_\_next\_\_ methods.

Iterators are everywhere in Python. They are elegantly implemented within for loops, comprehensions, generators etc. but hidden in plain sight. Iterator in Python is simply an [object](https://www.programiz.com/python-programming/class) that can be iterated upon. An object which will return data, one element at a time.

Technically speaking, Python **iterator object** must implement two special methods, \_\_iter\_\_() and \_\_next\_\_(), collectively called the **iterator protocol**.

An object is called **iterable** if we can get an iterator from it. Most of built-in containers in Python like: [list](https://www.programiz.com/python-programming/list), [tuple](https://www.programiz.com/python-programming/tuple), [string](https://www.programiz.com/python-programming/string) etc. are iterables. The iter() function (which in turn calls the \_\_iter\_\_() method) returns an iterator from them.

## Iterating Through an Iterator in Python. We use the next() function to manually iterate through all the items of an iterator. When we reach the end and there is no more data to be returned, it will raise StopIteration. Following is an example.



A more elegant way of automatically iterating is by using the [for loop](https://www.programiz.com/python-programming/for-loop). Using this, we can iterate over any object that can return an iterator, for example list, string, file etc.

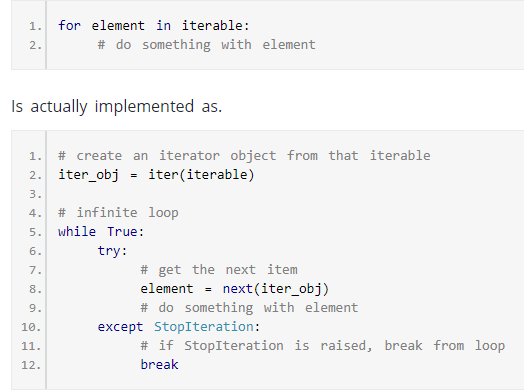
1. >>> for element in my\_list:
2. ... print(element)
3. ...
4. 4
5. 7
6. 0
7. 3

## 

## **How for loop actually works?**

As we see in the above example, the for loop was able to iterate automatically through the list.

In fact the for loop can iterate over any iterable. Let's take a closer look at how the forloop is actually implemented in Python.



So internally, the for loop creates an iterator object, iter\_obj by calling iter() on the iterable.

Ironically, this for loop is actually an infinite [while loop](https://www.programiz.com/python-programming/while-loop). Inside the loop, it calls next() to get the next element and executes the body of the forloop with this value. After all the items exhaust, StopIteration is raised which is internally caught and the loop ends. Note that any other kind of exception will pass through.

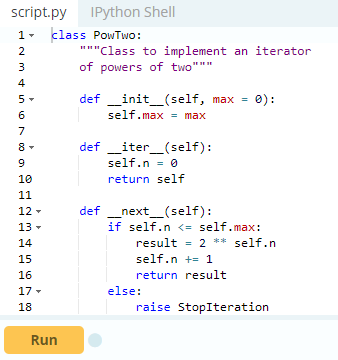
## **Building Your Own Iterator in Python**

Building an iterator from scratch is easy in Python. We just have to implement the methods \_\_iter\_\_() and \_\_next\_\_().

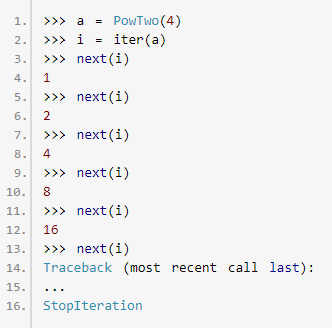
The \_\_iter\_\_() method returns the iterator object itself. If required, some initialization can be performed.

The \_\_next\_\_() method must return the next item in the sequence. On reaching the end, and in subsequent calls, it must raise StopIteration.

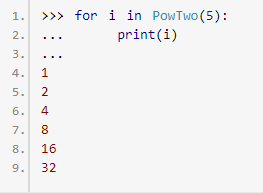
Here, we show an example that will give us next power of 2 in each iteration. Power exponent starts from zero up to a user set number.



Now we can create an iterator and iterate through it as follows.



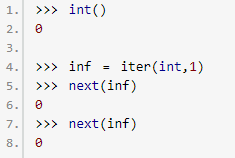
We can also use a for loop to iterate over our iterator class.



## **Python Infinite Iterators**

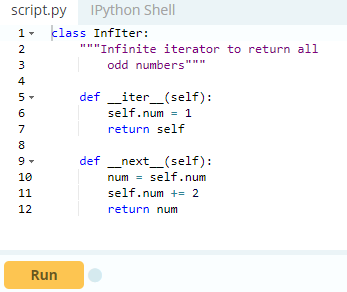
It is not necessary that the item in an iterator object has to exhaust. There can be infinite iterators (which never ends). We must be careful when handling such iterator. Here is a simple example to demonstrate infinite iterators.

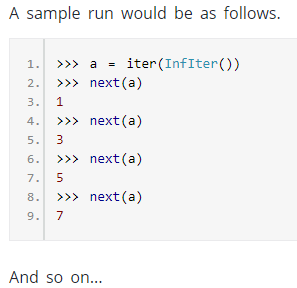
The [built-in function](https://www.programiz.com/python-programming/built-in-function) iter() can be called with two arguments where the first argument must be a callable object (function) and second is the sentinel. The iterator calls this function until the returned value is equal to the sentinel.



We can see that the int() function always returns 0. So passing it as iter(int,1) will return an iterator that calls int() until the returned value equals 1. This never happens and we get an infinite iterator.

We can also built our own infinite iterators. The following iterator will, theoretically, return all the odd numbers.





Be careful to include a terminating condition, when iterating over these type of infinite iterators. The advantage of using iterators is that they save resources. Like shown above, we could get all the odd numbers without storing the entire number system in memory. We can have infinite items (theoretically) in finite memory. Iterator also makes our code look cool.

# Python Generators

There is a lot of overhead in building an [iterator in Python](https://www.programiz.com/python-programming/iterator); we have to implement a class with \_\_iter\_\_() and \_\_next\_\_() method, keep track of internal states, raise Stop Iteration when there was no values to be returned etc.

This is both lengthy and counter intuitive. Generator comes into rescue in such situations. Python generators are a simple way of creating iterators. All the overhead we mentioned above are automatically handled by generators in Python. Simply speaking, a generator is a function that returns an object (iterator) which we can iterate over (one value at a time).

## **How to create a generator in Python?**

It is fairly simple to create a generator in Python. It is as easy as defining a normal function with yield statement instead of a return statement. If a function contains at least one yield statement (it may contain other yield or return statements), it becomes a generator function. Both yield and return will return some value from a function.

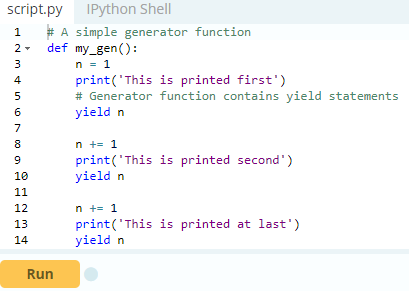
The difference is that, while a return statement terminates a function entirely, yield statement pauses the function saving all its states and later continues from there on successive calls.

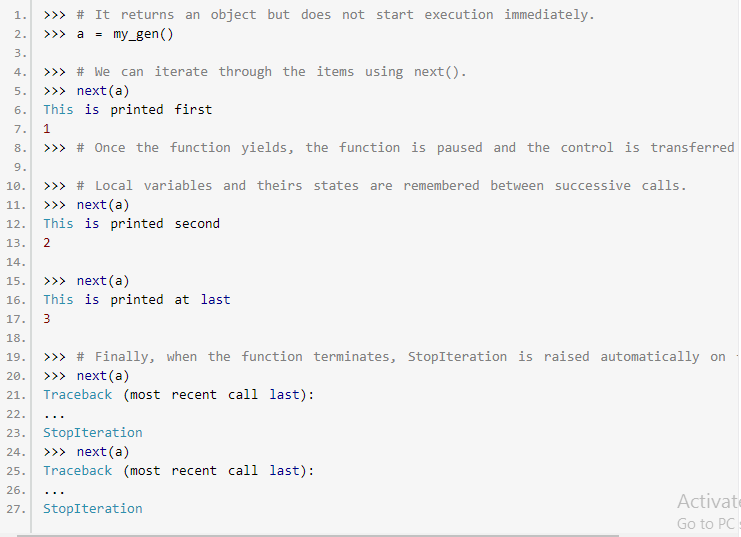
## **Differences between Generator function and a Normal function**

Here is how a generator function differs from a [normal function](https://www.programiz.com/python-programming/function).

* Generator function contains one or more yield statement.
* When called, it returns an object (iterator) but does not start execution immediately.
* Methods like \_\_iter\_\_() and \_\_next\_\_() are implemented automatically. So we can iterate through the items using next().
* Once the function yields, the function is paused and the control is transferred to the caller.
* Local variables and their states are remembered between successive calls.
* Finally, when the function terminates, Stop Iteration is raised automatically on further calls.

Here is an example to illustrate all of the points stated above. We have a generator function named my\_gen() with several yield statements.

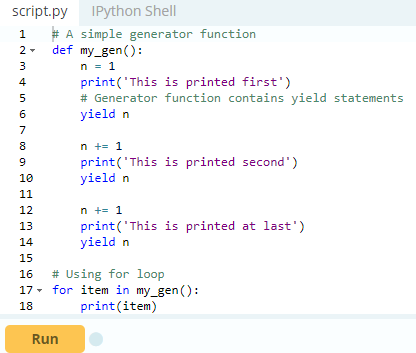


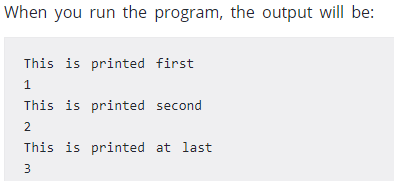


One interesting thing to note in the above example is that, the value of variable n is remembered between each call. Unlike normal functions, the local variables are not destroyed when the function yields. Furthermore, the generator object can be iterated only once. To restart the process we need to create another generator object using something like a = my\_gen().

**Note:**One final thing to note is that we can use generators with [for loops](https://www.programiz.com/python-programming/for-loop) directly.

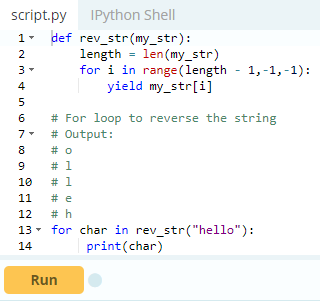
This is because, a for loop takes an iterator and iterates over it using next() function. It automatically ends when Stop Iteration is raised. Check here to [know how a for loop is actually implemented in Python](https://www.programiz.com/python-programming/iterator#for-loop-working).





## **Python Generators with a Loop**

The above example is of less use and we studied it just to get an idea of what was happening in the background. Normally, generator functions are implemented with a loop having a suitable terminating condition. Let's take an example of a generator that reverses a string.



In this example, we use range () function to get the index in reverse order using the for loop.

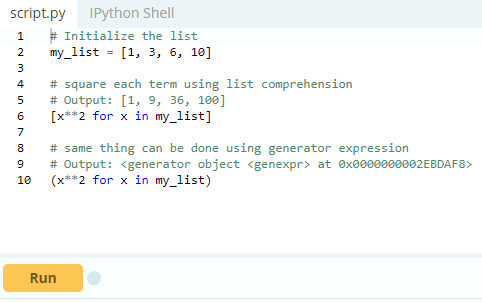
It turns out that this generator function not only works with string, but also with other kind of iterables like [list](https://www.programiz.com/python-programming/list), [tuple](https://www.programiz.com/python-programming/tuple) etc.

## **Python Generator Expression**

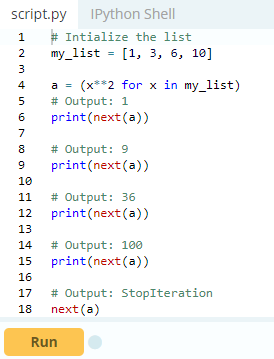
Simple generators can be easily created on the fly using generator expressions. It makes building generators easy. Same as lambda function creates an [anonymous function](https://www.programiz.com/python-programming/anonymous-function), generator expression creates an anonymous generator function.

The syntax for generator expression is similar to that of a [list comprehension in Python](https://www.programiz.com/python-programming/list#list-comprehension). But the square brackets are replaced with round parentheses. The major difference between a list comprehension and a generator expression is that while list comprehension produces the entire list, generator expression produces one item at a time.

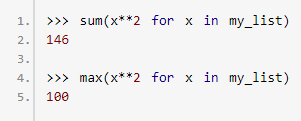
They are kind of lazy, producing items only when asked for. For this reason, a generator expression is much more memory efficient than an equivalent list comprehension.



We can see above that the generator expression did not produce the required result immediately. Instead, it returned a generator object with produces items on demand.



Generator expression can be used inside functions. When used in such a way, the round parentheses can be dropped.

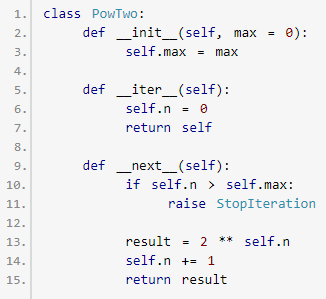


## **Why generators are used in Python?**

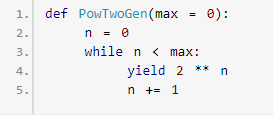
There are several reasons which make generators an attractive implementation to go for.

### **1. Easy to Implement**

Generators can be implemented in a clear and concise way as compared to their iterator class counterpart. Following is an example to implement a sequence of power of 2's using iterator class.



This was lengthy. Now lets do the same using a generator function.



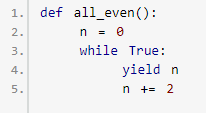
Since, generators keep track of details automatically, it was concise and much cleaner in implementation.

### **2. Memory Efficient**

A normal function to return a sequence will create the entire sequence in memory before returning the result. This is an overkill if the number of items in the sequence is very large. Generator implementation of such sequence is memory friendly and is preferred since it only produces one item at a time.

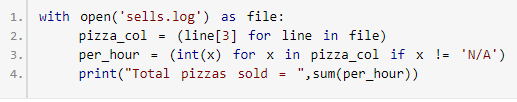
### **3. Represent Infinite Stream**

Generators are excellent medium to represent an infinite stream of data. Infinite streams cannot be stored in memory and since generators produce only one item at a time, it can represent infinite stream of data. The following example can generate all the even numbers (at least in theory).



### **4. Pipelining Generators**

Generators can be used to pipeline a series of operations. This is best illustrated using an example. Suppose we have a log file from a famous fast food chain. The log file has a column (4th column) that keeps track of the number of pizza sold every hour and we want to sum it to find the total pizzas sold in 5 years. Assume everything is in string and numbers that are not available are marked as 'N/A'. A generator implementation of this could be as follows.

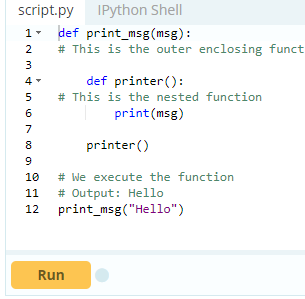


This pipelining is efficient and easy to read (and yes, a lot cooler!).

# Python Closures

## Nonlocal variable in a nested function. Before getting into what a closure is, we have to first understand what a nested function and nonlocal variable is. A function defined inside another function is called a nested function. Nested functions can access variables of the enclosing scope.

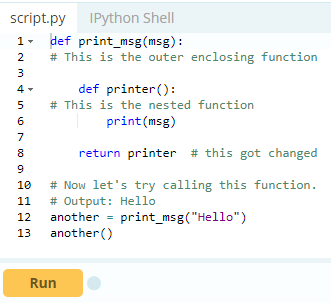
In Python, these non-local variables are read only by default and we must declare them explicitly as non-local (using [nonlocal keyword](https://www.programiz.com/python-programming/keyword-list#nonlocal)) in order to modify them. Following is an example of a nested function accessing a non-local variable.



We can see that the nested function printer() was able to access the non-local variable msgof the enclosing function.

## **Defining a Closure Function**

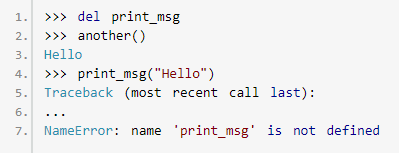
In the example above, what would happen if the last line of the function print\_msg()returned the printer() function instead of calling it? This means the function was defined as follows.

  
That's unusual.

The print\_msg() function was called with the string "Hello" and the returned function was bound to the name another. On calling another(), the message was still remembered although we had already finished executing the print\_msg() function. This technique by which some data ("Hello") gets attached to the code is called **closure in Python**.

This value in the enclosing scope is remembered even when the variable goes out of scope or the function itself is removed from the current namespace.

Try running the following in the Python shell to see the output.



## **When do we have a closure?**

As seen from the above example, we have a closure in Python when a nested function references a value in its enclosing scope. The criteria that must be met to create closure in Python are summarized in the following points.

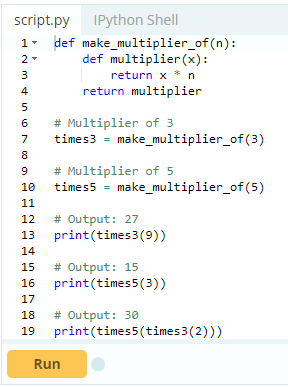
* We must have a nested function (function inside a function).
* The nested function must refer to a value defined in the enclosing function.
* The enclosing function must return the nested function.

## **When to use closures?**

So what are closures good for?

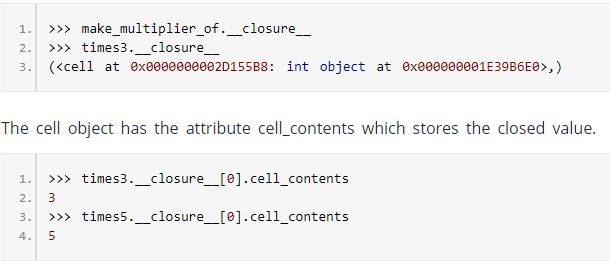
Closures can avoid the use of global values and provides some form of data hiding. It can also provide an object oriented solution to the problem.

When there are few methods (one method in most cases) to be implemented in a class, closures can provide an alternate and more elegant solutions. But when the number of attributes and methods get larger, better implement a class. Here is a simple example where a closure might be more preferable than defining a class and making objects. But the preference is all yours.



[Decorators in Python](https://www.programiz.com/python-programming/decorator) make an extensive use of closures as well. On a concluding note, it is good to point out that the values that get enclosed in the closure function can be found out.

All function objects have a \_\_closure\_\_ attribute that returns a tuple of cell objects if it is a closure function. Referring to the example above, we know times3 and times5 are closure functions.



# Python Decorators

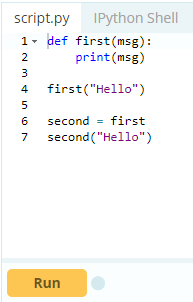
A decorator takes in a function, adds some functionality and returns it. In this article, you will learn how you can create a decorator and why you should use it.

Python has an interesting feature called **decorators** to add functionality to an existing code.

This is also called **metaprogramming** as a part of the program tries to modify another part of the program at compile time.

## **Prerequisites for learning decorators**

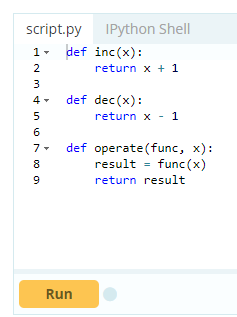
In order to understand about decorators, we must first know a few basic things in Python. We must be comfortable with the fact that, everything in Python (Yes! Even classes), are [objects](https://www.programiz.com/python-programming/class). Names that we define are simply identifiers bound to these objects. [Functions](https://www.programiz.com/python-programming/function) are no exceptions, they are objects too (with attributes). Various different names can be bound to the same function object. Here is an example.



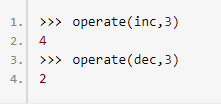
When you run the code, both functions first and second gives same output. Here, the names first and second refer to the same function object.

Now things start getting weirder.

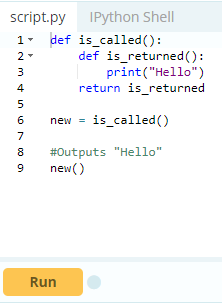
Functions can be passed as arguments to another function. If you have used functions like map, filter and reduce in Python, then you already know about this. Such function that take other functions as arguments are also called **higher order functions**. Here is an example of such a function.



We invoke the function as follows.



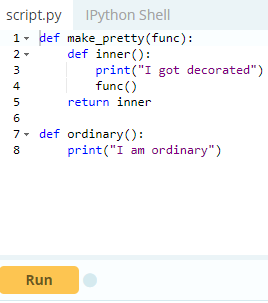
Furthermore, a function can return another function.



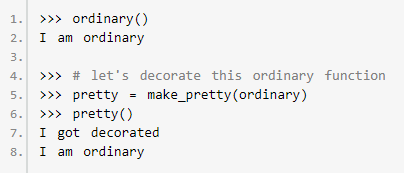
Here, is\_returned() is a nested function which is defined and returned, each time we call is\_called(). Finally, we must know about [closures in Python](https://www.programiz.com/python-programming/closure).

## **Getting back to Decorators**

Functions and methods are called **callable** as they can be called. In fact, any object which implements the special method \_\_call\_\_() is termed callable. So, in the most basic sense, a decorator is a callable that returns a callable. Basically, a decorator takes in a function, adds some functionality and returns it.



When you run the following codes in shell,



In the example shown above, make\_pretty() is a decorator. In the assignment step.

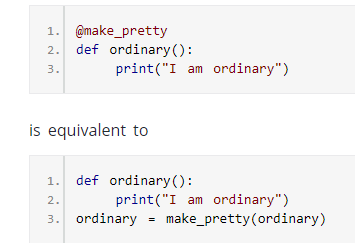


The function ordinary() got decorated and the returned function was given the name pretty. We can see that the decorator function added some new functionality to the original function. This is similar to packing a gift. The decorator acts as a wrapper. The nature of the object that got decorated (actual gift inside) does not alter. But now, it looks pretty (since it got decorated).

Generally, we decorate a function and reassign it as,



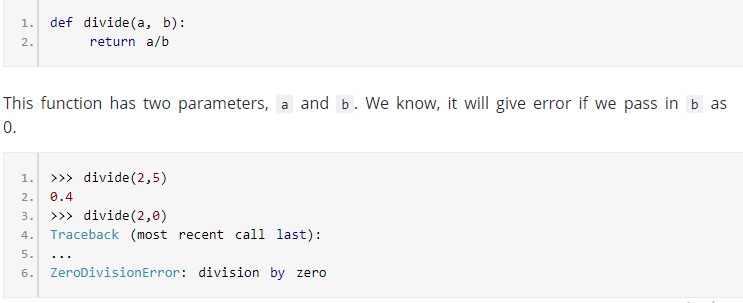
This is a common construct and for this reason, Python has a syntax to simplify this. We can use the @ symbol along with the name of the decorator function and place it above the definition of the function to be decorated. For example,



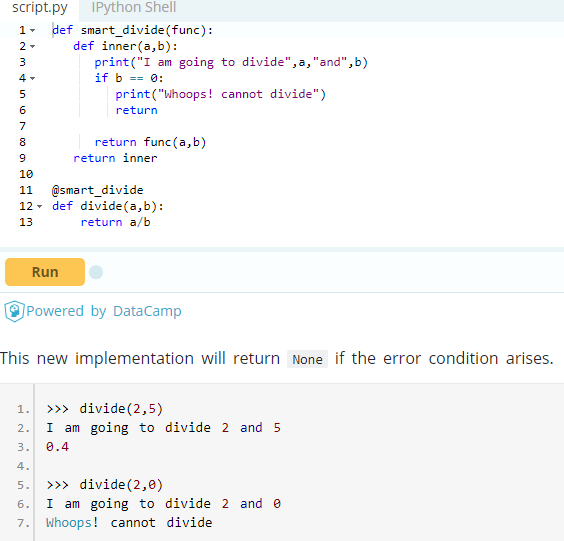
This is just a syntactic sugar to implement decorators.

## **Decorating Functions with Parameters**

The above decorator was simple and it only worked with functions that did not have any parameters. What if we had functions that took in parameters like below?



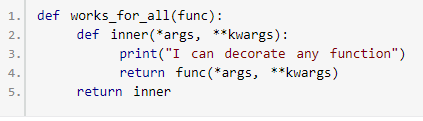
Now let's make a decorator to check for this case that will cause the error.



In this manner we can decorate functions that take parameters.

A keen observer will notice that parameters of the nested inner() function inside the decorator is same as the parameters of functions it decorates. Taking this into account, now we can make general decorators that work with any number of parameter.

In Python, this magic is done as function(\*args, \*\*kwargs). In this way, args will be the [tuple](https://www.programiz.com/python-programming/tuple) of positional arguments and kwargs will be the [dictionary](https://www.programiz.com/python-programming/dictionary) of keyword arguments. An example of such decorator will be.



## 

## **Chaining Decorators in Python**

Multiple decorators can be chained in Python. This is to say, a function can be decorated multiple times with different (or same) decorators. We simply place the decorators above the desired function.

# 

# 

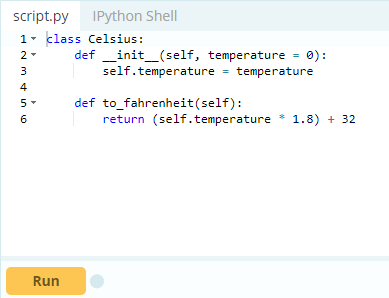
# Python @property

**You will learn about Python @property; pythonic way to use getters and setters.**

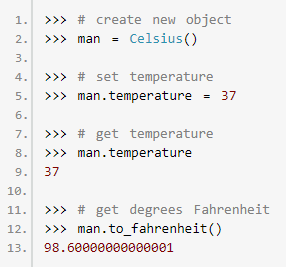
Python has a great concept called property which makes the life of an object oriented programmer much simpler. Before defining and going into details of what @property is, let us first build an intuition on why it would be needed in the first place.

## An Example To Begin With

Let us assume that you decide to [make a class](https://www.programiz.com/python-programming/class) that could store the temperature in degree Celsius. It would also implement a method to convert the temperature into degree Fahrenheit. One way of doing this is as follows.



We could make objects out of this class and manipulate the attribute temperature as we wished. Try these on Python shell.



The extra decimal places when converting into Fahrenheit is due to the floating point arithmetic error (try 1.1 + 2.2 in the Python interpreter).

Whenever we assign or retrieve any object attribute like temperature, as show above, Python searches it in the object's \_\_dict\_\_ dictionary.



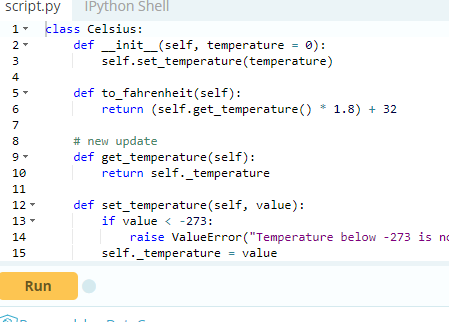
Therefore, man.temperature internally becomes man.\_\_dict\_\_['temperature'].

Now, let's further assume that our class got popular among clients and they started using it in their programs. They did all kinds of assignments to the object.

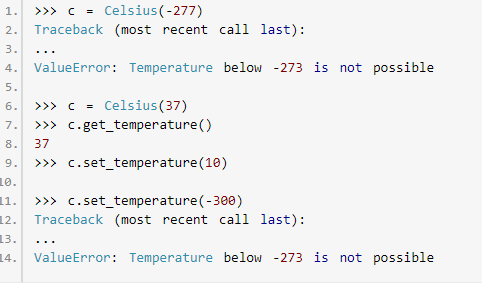
One fateful day, a trusted client came to us and suggested that temperatures cannot go below -273 degree Celsius (students of thermodynamics might argue that it's actually -273.15), also called the absolute zero. He further asked us to implement this value constraint. Being a company that strive for customer satisfaction, we happily heeded the suggestion and released version 1.01 (an upgrade of our existing class).

## **Using Getters and Setters**

An obvious solution to the above constraint will be to hide the attribute temperature (make it private) and define new getter and setter interfaces to manipulate it. This can be done as follows.

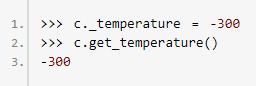


We can see above that new methods get\_temperature() and set\_temperature() were defined and furthermore, temperature was replaced with \_temperature. An underscore (\_) at the beginning is used to denote private variables in Python.



This update successfully implemented the new restriction. We are no longer allowed to set temperature below -273.

Please note that private variables don't exist in Python. There are simply norms to be followed. The language itself don't apply any restrictions.

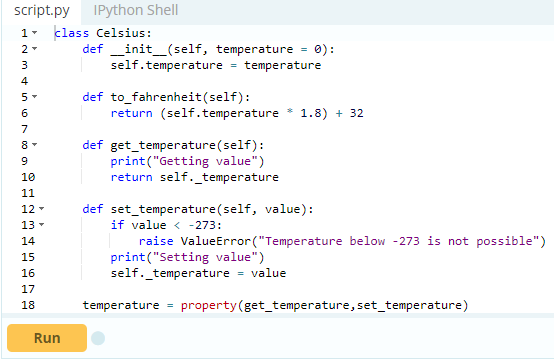


But this is not of great concern. The big problem with the above update is that, all the clients who implemented our previous class in their program have to modify their code from obj.temperature to obj.get\_temperature() and all assignments like obj.temperature = val to obj.set\_temperature(val).

This refactoring can cause headaches to the clients with hundreds of thousands of lines of codes. All in all, our new update was not backward compatible. This is where property comes to rescue.

## **The Power of @property**

The pythonic way to deal with the above problem is to use property. Here is how we could have achieved it.



And, issue the following code in shell once you run it.

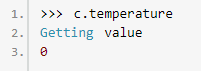
>>> c = Celsius()

We added a print() function inside get\_temperature() and set\_temperature() to clearly observe that they are being executed. The last line of the code, makes a property object temperature. Simply put, property attaches some code (get\_temperature and set\_temperature) to the member attribute accesses (temperature).

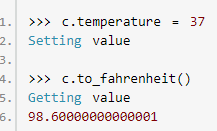
Any code that retrieves the value of temperature will automatically call get\_temperature()instead of a dictionary (\_\_dict\_\_) look-up. Similarly, any code that assigns a value to temperature will automatically call set\_temperature(). This is one cool feature in Python. We can see above that set\_temperature() was called even when we created an object.

**Can you guess why?**

The reason is that when an object is created, \_\_init\_\_() method gets called. This method has the line self.temperature = temperature. This assignment automatically called set\_temperature().



Similarly, any access like c.temperature automatically calls get\_temperature(). This is what property does. Here are a few more examples.



By using property, we can see that, we modified our class and implemented the value constraint without any change required to the client code. Thus our implementation was backward compatible and everybody is happy.

Finally note that, the actual temperature value is stored in the private variable \_temperature. The attribute temperature is a property object which provides interface to this private variable.

## **Digging Deeper into Property**

In Python, property() is a built-in function that creates and returns a property object. The signature of this function is



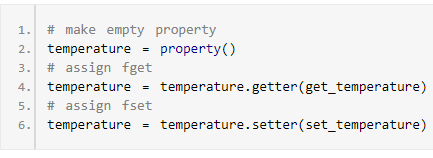
where, fget is function to get value of the attribute, fset is function to set value of the attribute, fdel is function to delete the attribute and doc is a string (like a comment). As seen from the implementation, these function arguments are optional. So, a property object can simply be created as follows.



A property object has three methods, getter(), setter(), and deleter() to specify fget, fset and fdel at a later point. This means, the line

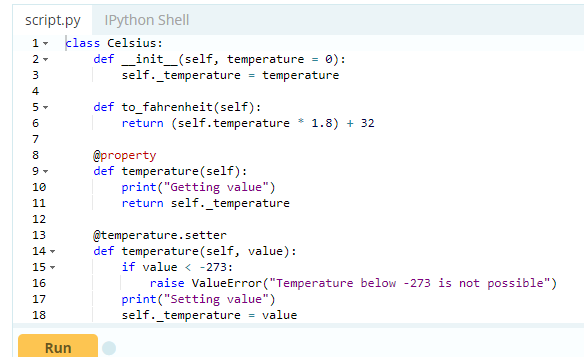


could have been broken down as



These two pieces of codes are equivalent. Programmers familiar with [decorators in Python](https://www.programiz.com/python-programming/decorator) can recognize that the above construct can be implemented as decorators.

We can further go on and not define names get\_temperature and set\_temperature as they are unnecessary and pollute the class namespace. For this, we reuse the name temperature while defining our getter and setter functions. This is how it can be done.



The above implementation is both, simple and recommended way to make properties. You will most likely encounter these types of constructs when looking for property in Python.

# Python Shallow Copy and Deep Copy

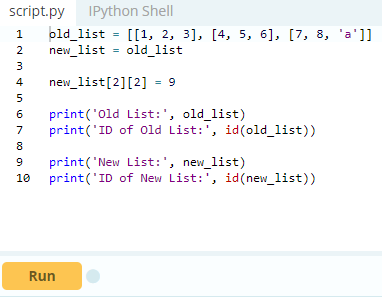
In this article, you’ll learn about shallow copy and deep copy in Python with the help of examples.

## Copy an Object in Python

In Python, we use = operator to create a copy of an object. You may think that this creates a new object; it doesn't. It only creates a new variable that shares the reference of the original object.

Let's take an example where we create a list named old\_list and pass an object reference to new\_list using = operator.

### **Example 1: Copy using = operator**



When we run above program, the output will be:

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

ID of Old List: 140673303268168

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

ID of New List: 140673303268168

As you can see from the output both variables old\_list and new\_list shares the same id i.e 140673303268168.

So, if you want to modify any values in new\_list or old\_list, the change is visible in both.

Essentially, sometimes you may want to have the original values unchanged and only modify the new values or vice versa. In Python, there are two ways to create copies:

1. Shallow Copy
2. Deep Copy

To make these copy work, we use the copy module.

## **Copy Module**

We use the copy module of Python for shallow and deep copy operations. Suppose, you need to copy the compound list say x. For example:

import copy

copy.copy(x)

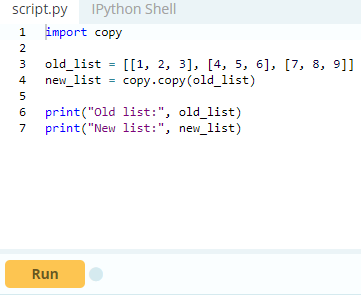
copy.deepcopy(x)

Here, the copy() return a shallow copy of x. Similarly, deepcopy() return a deep copy of x.

## **Shallow Copy**

A shallow copy creates a new object which stores the reference of the original elements. So, a shallow copy doesn't create a copy of nested objects, instead it just copies the reference of nested objects. This means, a copy process does not recurse or create copies of nested objects itself.

### **Example 2: Create a copy using shallow copy**



When we run the program , the output will be:

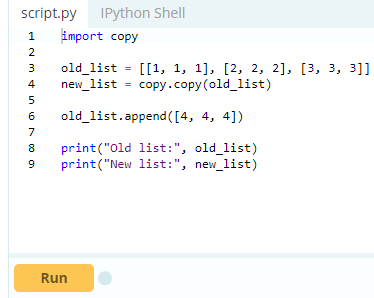
Old list: [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

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

In above program, we created a nested list and then shallow copy it using copy() method.

This means it will create new and independent object with same content. To verify this, we print the both old\_list and new\_list. To confirm that new\_list is different from old\_list, we try to add new nested object to original and check it.

### **Example 3: Adding [4, 4, 4] to old\_list, using shallow copy**



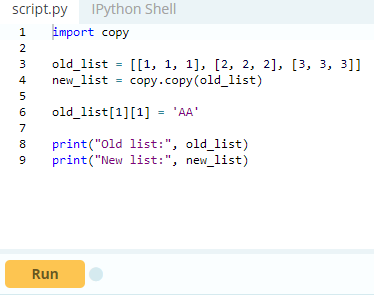
When we run the program, it will output:

Old list: [[1, 1, 1], [2, 2, 2], [3, 3, 3], [4, 4, 4]]

New list: [[1, 1, 1], [2, 2, 2], [3, 3, 3]]

In the above program, we created a shallow copy of old\_list. The new\_list contains references to original nested objects stored in old\_list. Then we add the new list i.e [4, 4, 4] into old\_list. This new sublist was not copied in new\_list. However, when you change any nested objects in old\_list, the changes appear in new\_list.

**Example 4: Adding new nested object using Shallow copy**



When we run the program, it will output:

Old list: [[1, 1, 1], [2, 'AA', 2], [3, 3, 3]]

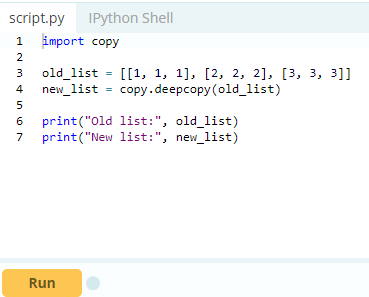
New list: [[1, 1, 1], [2, 'AA', 2], [3, 3, 3]]

In the above program, we made changes to old\_list i.e old\_list[1][1] = 'AA'. Both sublists of old\_list and new\_list at index [1][1] were modified. This is because, both lists share the reference of same nested objects.

## **Deep Copy**

A deep copy creates a new object and recursively adds the copies of nested objects present in the original elements. Let’s continue with example 2. However, we are going to create deep copy using deepcopy()function present in copy module. The deep copy creates independent copy of original object and all its nested objects.

### **Example 5: Copying a list using deepcopy()**



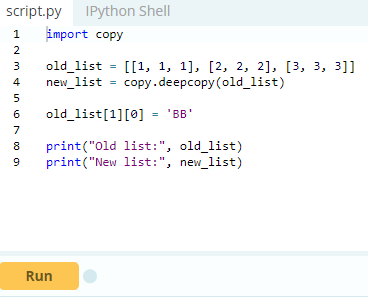
When we run the program, it will output:

Old list: [[1, 1, 1], [2, 2, 2], [3, 3, 3]]

New list: [[1, 1, 1], [2, 2, 2], [3, 3, 3]]

In the above program, we use deepcopy() function to create copy which looks similar. However, if you make changes to any nested objects in original object old\_list, you’ll see no changes to the copy new\_list.

### **Example 6: Adding a new nested object in the list using Deep copy**



When we run the program, it will output:

Old list: [[1, 1, 1], ['BB', 2, 2], [3, 3, 3]]

New list: [[1, 1, 1], [2, 2, 2], [3, 3, 3]]

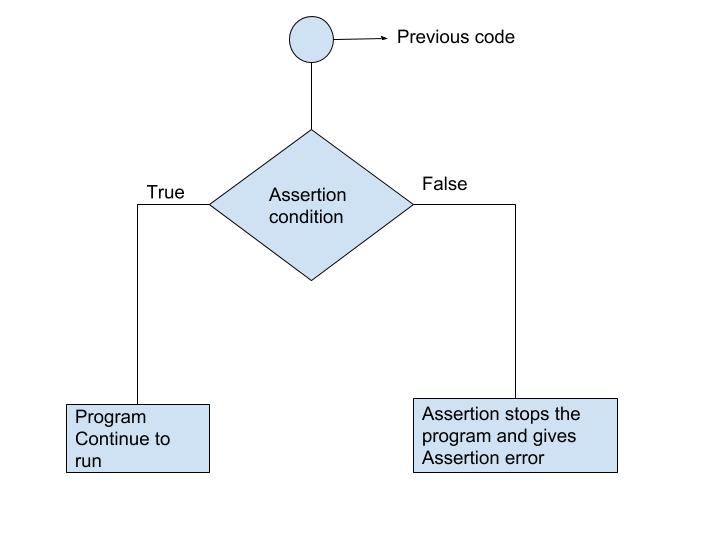
In the above program, when we assign a new value to old\_list, we can see only the old\_list is modified. This means, both the old\_list and the new\_list are independent. This is because the old\_list was recursively copied, which is true for all its nested objects.

# Python Assert Statement

Assertions are statements that assert or state a fact confidently in your program. For example, while writing a division function, you're confident the divisor shouldn't be zero, you assert divisor is not equal to zero.

Assertions are simply boolean expressions that checks if the conditions return true or not. If it is true, the program does nothing and move to the next line of code. However, if it's false, the program stops and throws an error.

It is also a debugging tool as it brings the program on halt as soon as any error is occurred and shows on which point of the program error has occurred. You can learn more about assertions in the article: [The benefits of programming with Assertions](http://pgbovine.net/programming-with-asserts.htm) We can be clear by looking at the flowchart below:



## **Python assert Statement**

Python has built-in assert statement to use assertion condition in the program. Assert statement has a condition or expression which is supposed to be always true. If the condition is false assert halts the program and gives an AssertionError.

#### **Syntax for using Assert in Python:**

assert <condition>

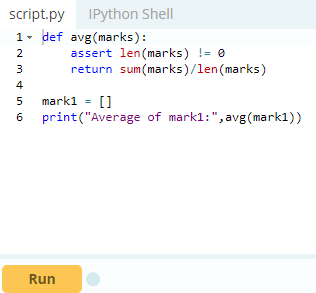
assert <condition>,<error message>

In Python we can use assert statement in two ways as mentioned above.

1. assert statement has a condition and if the condition is not satisfied the program will stop and give AssertionError.
2. assert statement can also have a condition and a optional error message. If the condition is not satisfied assert stops the program and gives AssertionError along with the error message.

Let's take an example, where we have a function which will calculate the average of the values passed by the user and the value should not be an empty list. We will use assertstatement to check the parameter and if the length is of the passed list is zero, program halts.

### **Example 1: Using assert without Error Message**

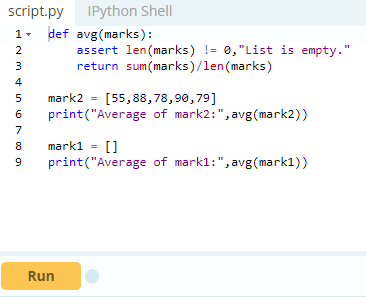


When we run the above program, the output will be:

AssertionError

We got an error as we passed an empty list mark1 to assert statement, the condition became false and assert stops the program and give AssertionError. Now let's pass another list which will satisfy the assert condition and see what will be our output.

### **Example 2: Using assert with error message**



When we run the above program, the output will be:

Average of mark2: 78.0

AssertionError: List is empty.

We passed a non-empty list mark2 and also an empty list mark1 to the avg() function and we got output for mark2 list but after that we got an error AssertionError: List is empty. The assert condition was satisfied by the mark2 list and program to continue to run. However, mark1 doesn't satisfy the condition and gives an AssertionError.

## **Key Points to Remember**

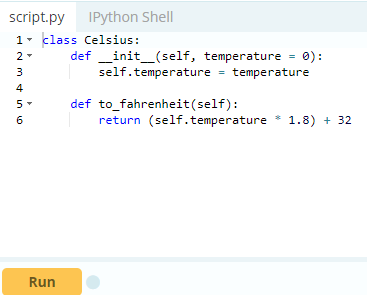
* Assertions are the condition or boolean expression which are always supposed to be true in the code.
* assert statement takes an expression and optional message.
* assert statement is used to check types, values of argument and the output of the function.
* assert statement is used as debugging tool as it halts the program at the point where an error occurs.

# Python @property

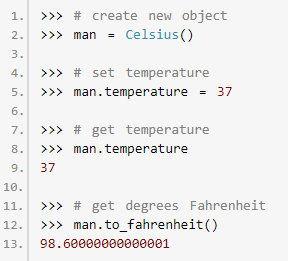
Python has a great concept called property which makes the life of an object oriented programmer much simpler. Before defining and going into details of what @property is, let us first build an intuition on why it would be needed in the first place.

## **An Example To Begin With**

Let us assume that you decide to [make a class](https://www.programiz.com/python-programming/class) that could store the temperature in degree Celsius. It would also implement a method to convert the temperature into degree Fahrenheit. One way of doing this is as follows.



We could make objects out of this class and manipulate the attribute temperature as we wished. Try these on Python shell.



The extra decimal places when converting into Fahrenheit is due to the floating point arithmetic error (try 1.1 + 2.2 in the Python interpreter).

Whenever we assign or retrieve any object attribute like temperature, as show above, Python searches it in the object's \_\_dict\_\_ dictionary.

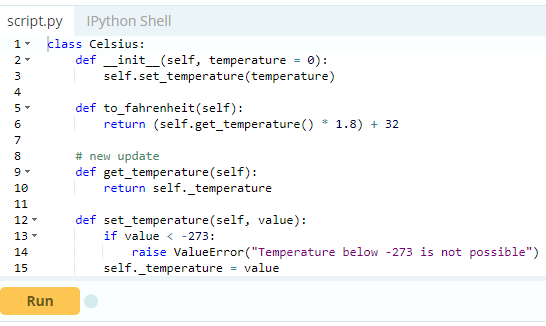
1. >>> man.\_\_dict\_\_
2. {'temperature': 37}

Therefore, man.temperature internally becomes man.\_\_dict\_\_['temperature']. Now, let's further assume that our class got popular among clients and they started using it in their programs. They did all kinds of assignments to the object.

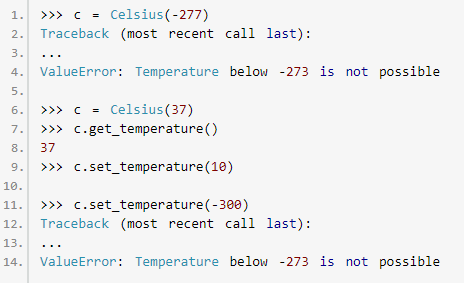
One fateful day, a trusted client came to us and suggested that temperatures cannot go below -273 degree Celsius (students of thermodynamics might argue that it's actually -273.15), also called the absolute zero. He further asked us to implement this value constraint. Being a company that strive for customer satisfaction, we happily heeded the suggestion and released version 1.01 (an upgrade of our existing class).

## **Using Getters and Setters**

An obvious solution to the above constraint will be to hide the attribute temperature (make it private) and define new getter and setter interfaces to manipulate it. This can be done as follows.

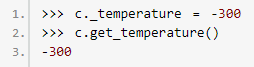


We can see above that new methods get\_temperature() and set\_temperature() were defined and furthermore, temperature was replaced with \_temperature. An underscore (\_) at the beginning is used to denote private variables in Python.



This update successfully implemented the new restriction. We are no longer allowed to set temperature below -273.

Please note that private variables don't exist in Python. There are simply norms to be followed. The language itself don't apply any restrictions.

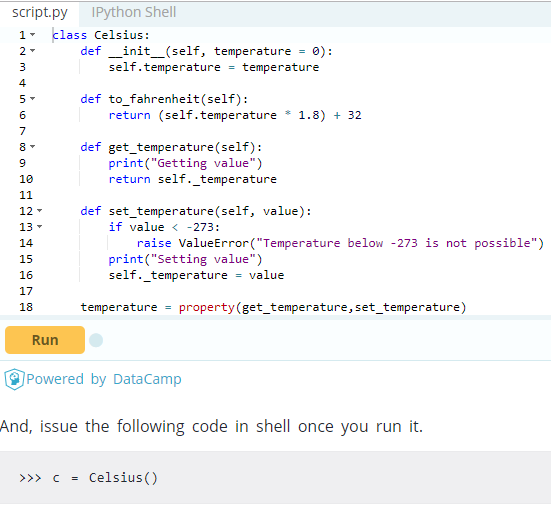


But this is not of great concern. The big problem with the above update is that, all the clients who implemented our previous class in their program have to modify their code from obj.temperature to obj.get\_temperature() and all assignments like obj.temperature = val to obj.set\_temperature(val).

This refactoring can cause headaches to the clients with hundreds of thousands of lines of codes. All in all, our new update was not backward compatible. This is where property comes to rescue.

## **The Power of @property**

The pythonic way to deal with the above problem is to use property. Here is how we could have achieved it.



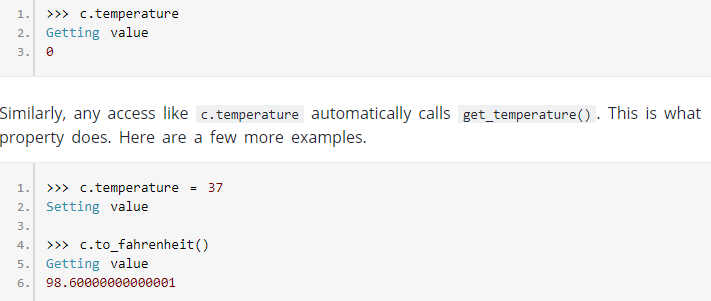
We added a print() function inside get\_temperature() and set\_temperature() to clearly observe that they are being executed.

The last line of the code, makes a property object temperature. Simply put, property attaches some code (get\_temperature and set\_temperature) to the member attribute accesses (temperature).

Any code that retrieves the value of temperature will automatically call get\_temperature()instead of a dictionary (\_\_dict\_\_) look-up. Similarly, any code that assigns a value to temperature will automatically call set\_temperature(). This is one cool feature in Python. We can see above that set\_temperature() was called even when we created an object.

**Can you guess why?**

The reason is that when an object is created, \_\_init\_\_() method gets called. This method has the line self.temperature = temperature. This assignment automatically called set\_temperature().



By using property, we can see that, we modified our class and implemented the value constraint without any change required to the client code. Thus our implementation was backward compatible and everybody is happy.

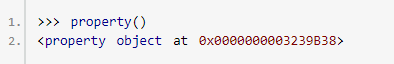
Finally note that, the actual temperature value is stored in the private variable \_temperature. The attribute temperature is a property object which provides interface to this private variable.

## **Digging Deeper into Property**

In Python, property() is a built-in function that creates and returns a property object. The signature of this function is



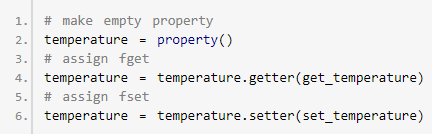
where, fget is function to get value of the attribute, fset is function to set value of the attribute, fdel is function to delete the attribute and doc is a string (like a comment). As seen from the implementation, these function arguments are optional. So, a property object can simply be created as follows.



A property object has three methods, getter(), setter(), and deleter() to specify fget, fset and fdel at a later point. This means, the line



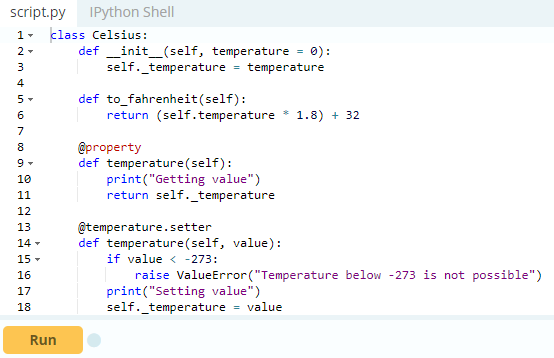
could have been broken down as



These two pieces of codes are equivalent.

Programmers familiar with [decorators in Python](https://www.programiz.com/python-programming/decorator) can recognize that the above construct can be implemented as decorators.

We can further go on and not define names get\_temperature and set\_temperature as they are unnecessary and pollute the class namespace. For this, we reuse the name temperature while defining our getter and setter functions. This is how it can be done.



The above implementation is both, simple and recommended way to make properties. You will most likely encounter these types of constructs when looking for property in Python.

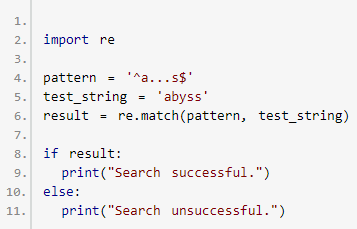
# *Python RegEx*

A **Re**gular **Ex**pression (RegEx) is a sequence of characters that defines a search pattern. For example,

1. ^a...s$

The above code defines a RegEx pattern. The pattern is: **any five letter string starting with**a**and ending with**s. A pattern defined using RegEx can be used to match against a string.

Python has a module named re to work with RegEx. Here's an example:



Here, we used re.match() function to search pattern within the test\_string. The method returns a match object if the search is successful. If not, it returns None.

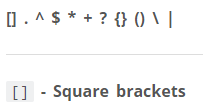
There are other several functions defined in the re module to work with RegEx. Before we explore that, let's learn about regular expressions themselves. If you already know the basics of RegEx, jump to [Python RegEx](https://www.programiz.com/python-programming/regex#python-regex).

## **Specify Pattern Using RegEx**

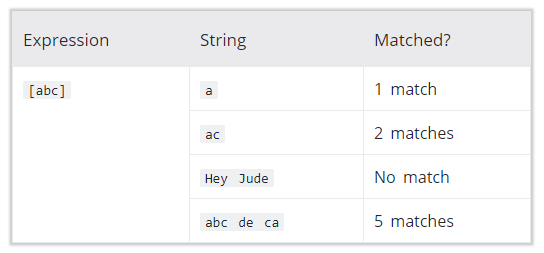
To specify regular expressions, metacharacters are used. In the above example, ^ and $ are metacharacters.

### **MetaCharacters**

Metacharacters are characters that are interpreted in a special way by a RegEx engine. Here's a list of metacharacters:



Square brackets specifies a set of characters you wish to match.



Here, [abc] will match if the string you are trying to match contains any of the a, b or c.

You can also specify a range of characters using - inside square brackets.

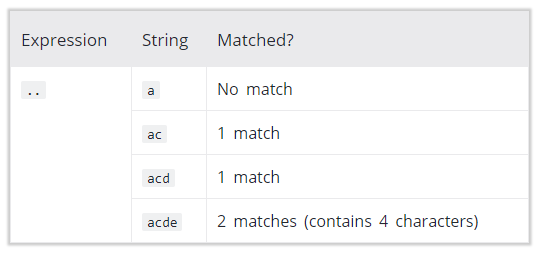
* [a-e] is the same as [abcde].
* [1-4] is the same as [1234].
* [0-39] is the same as [01239].

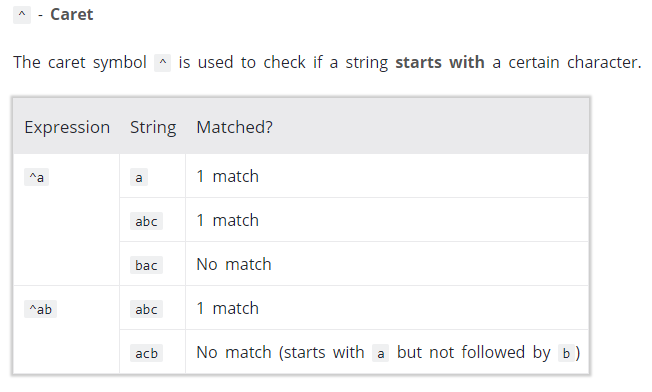
You can complement (invert) the character set by using caret ^ symbol at the start of a square-bracket.

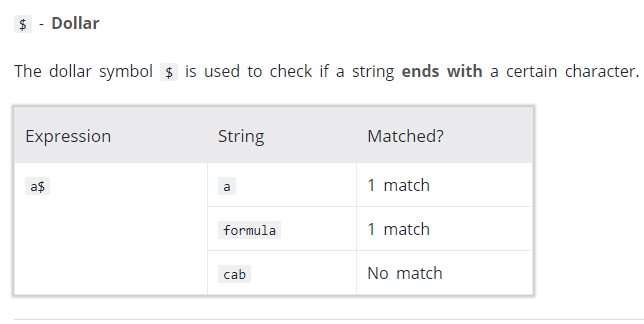
* [^abc] means any character except a or b or c.
* [^0-9] means any non-digit character.

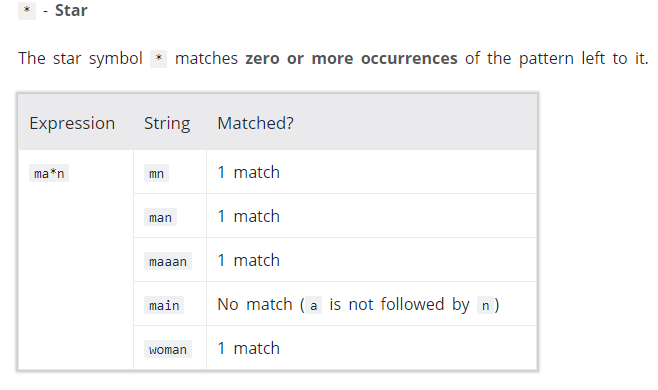
**. – Period**

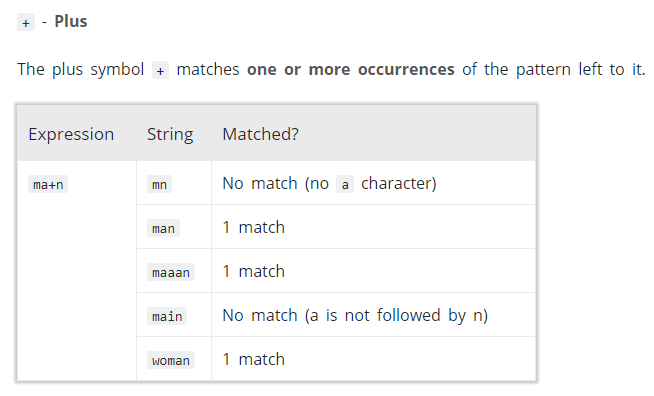
A period matches any single character (except newline '\n').

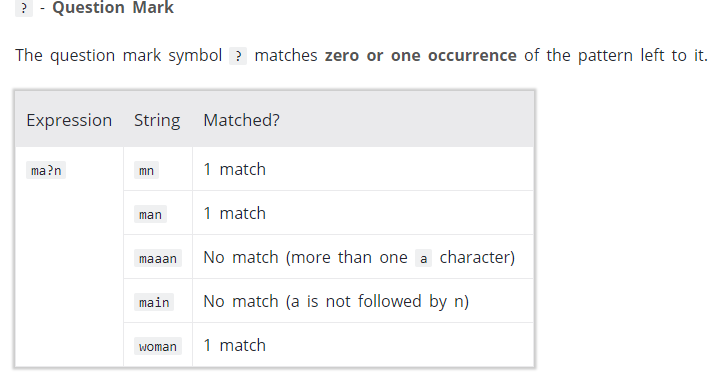


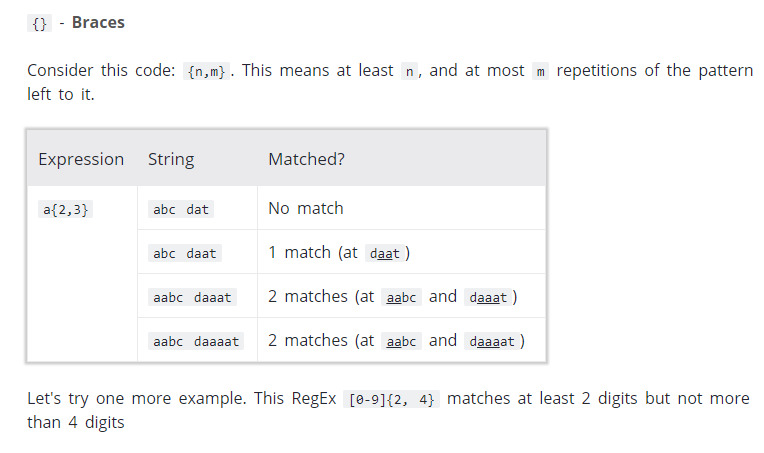


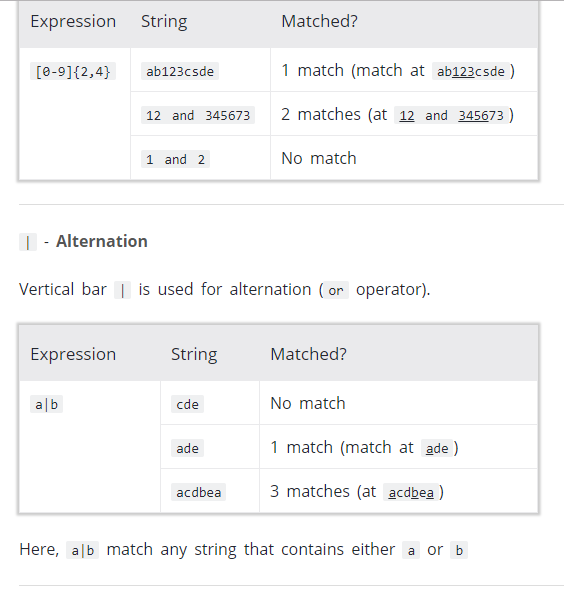


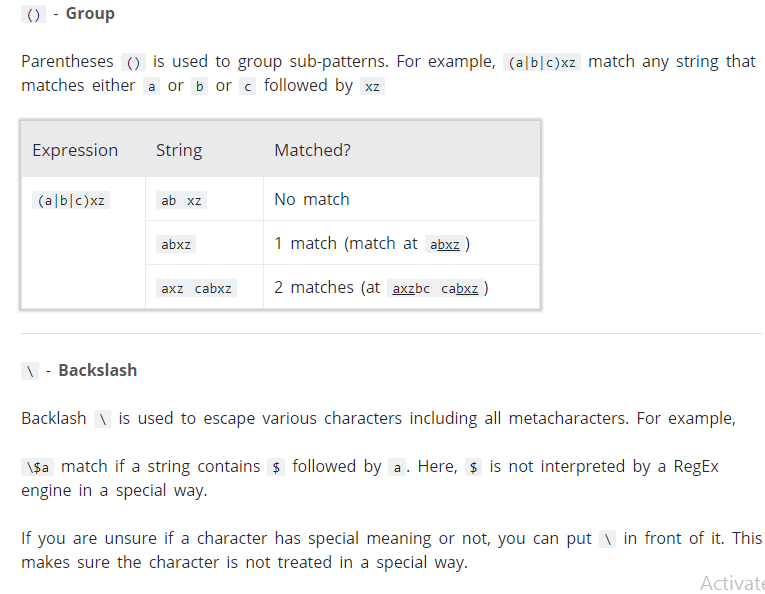


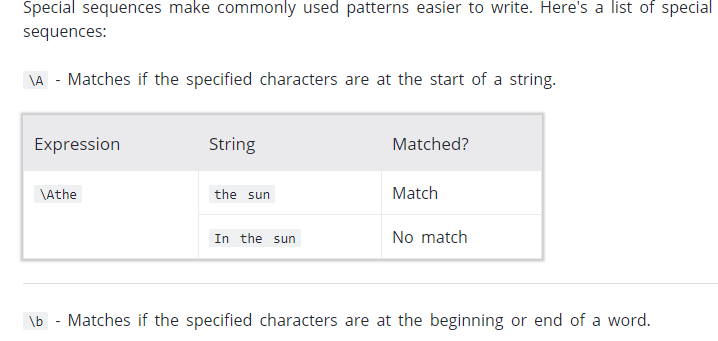




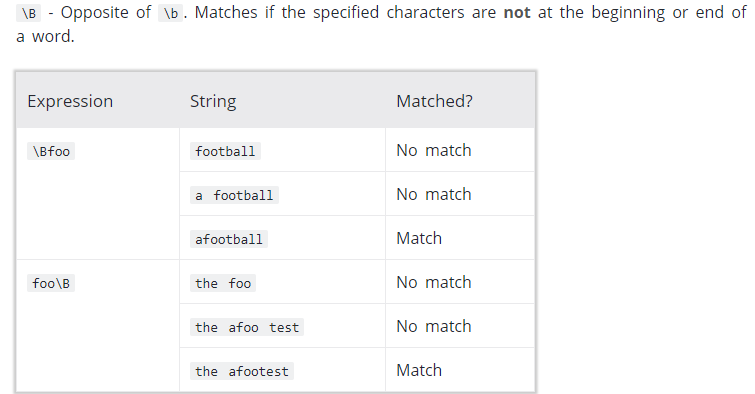


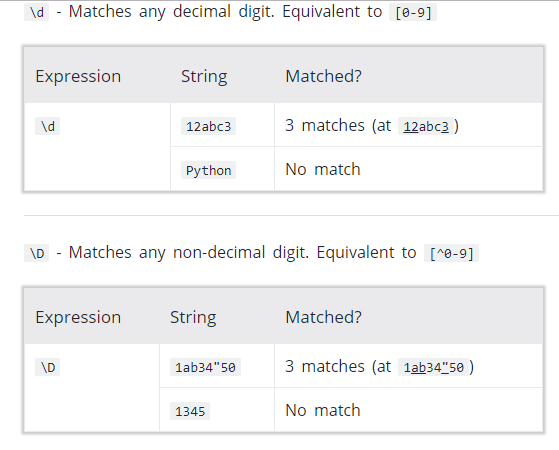


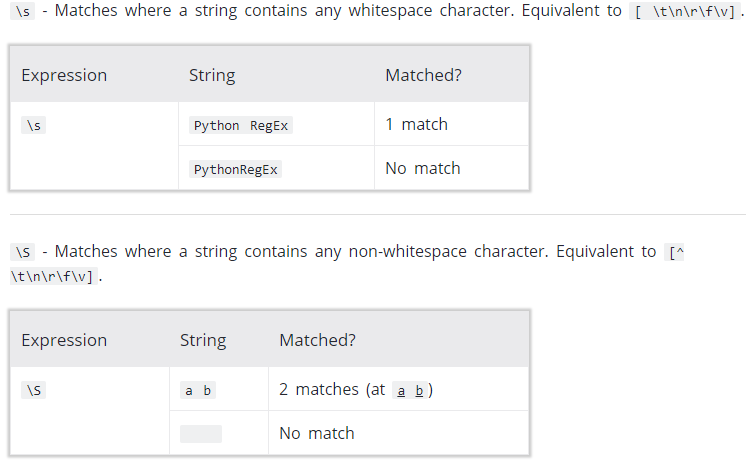


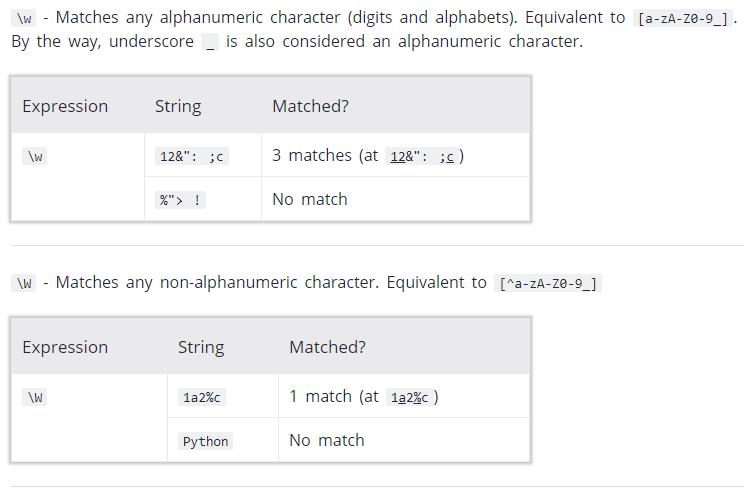


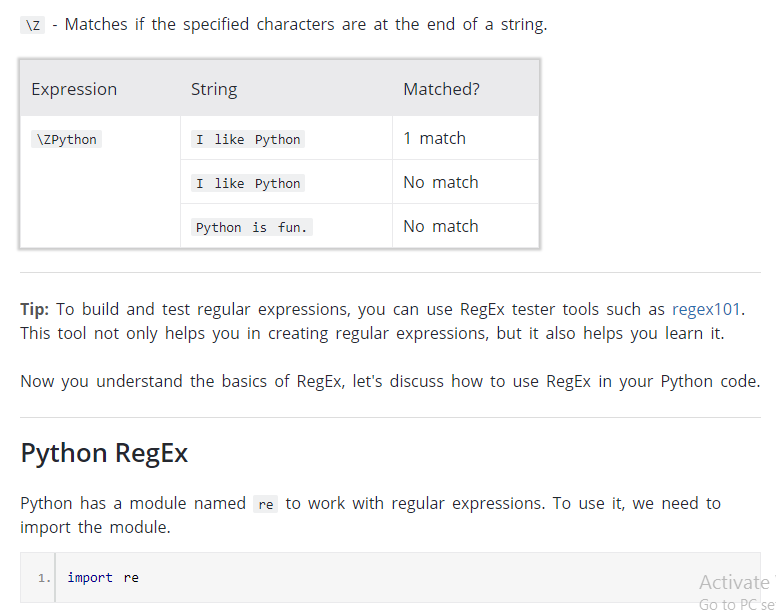


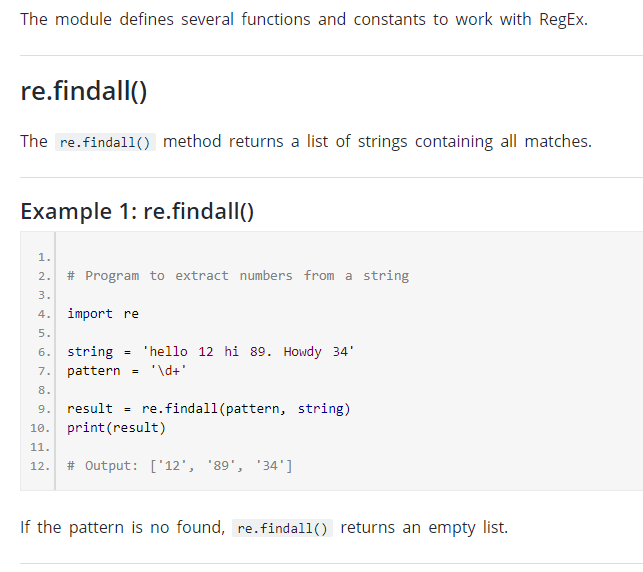


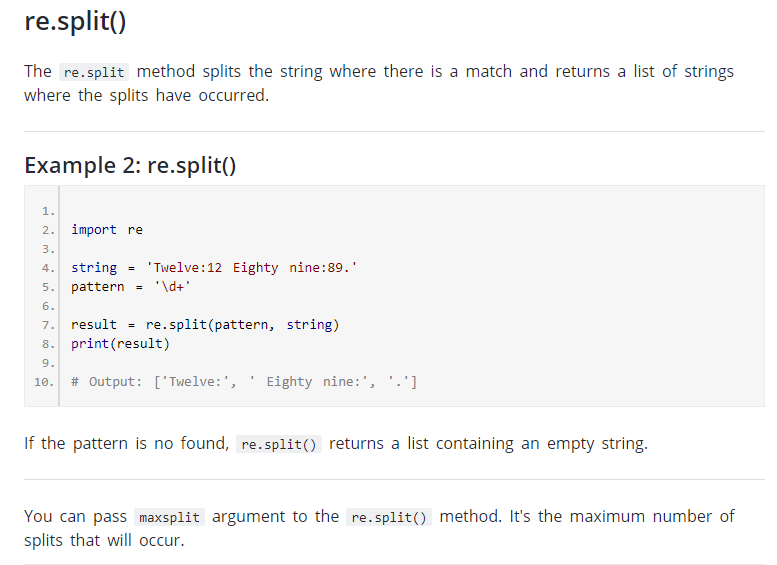




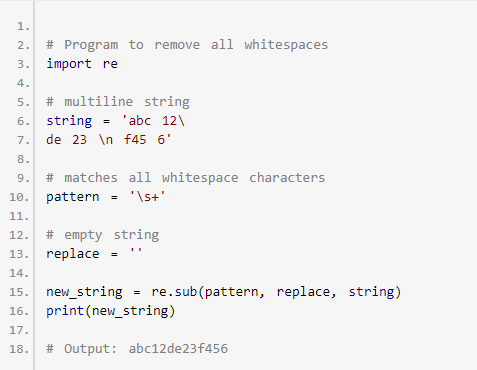






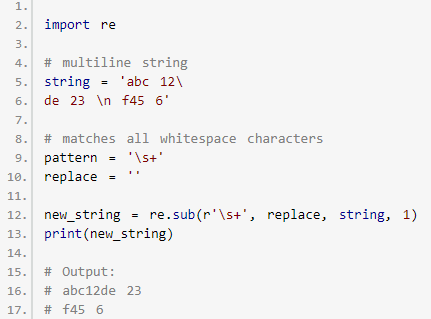






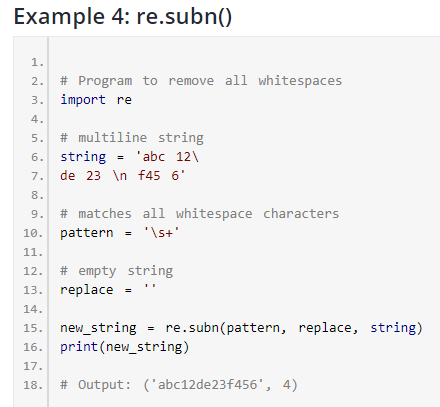
If the pattern is no found, re.sub() returns the original string

You can pass count as a fourth parameter to the re.sub() method. If omited, it results to 0. This will replace all occurrences.



## **re.subn()**

The re.subn() is similar to re.sub() expect it returns a tuple of 2 items containing the new string and the number of substitutions made.

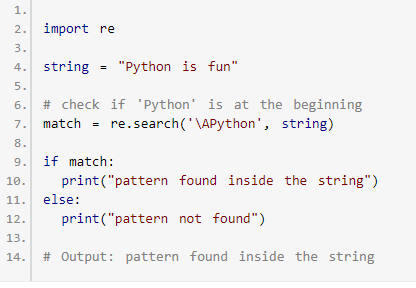


## **re.search()**

The re.search() method takes two arguments: a pattern and a string. The method looks for the first location where the RegEx pattern produces a match with the string.

If the search is successful, re.search() returns a match object; if not, it returns None.

1. match = re.search(pattern, str)



Here, match contains a match object

## **Match object**

You can get methods and attributes of a match object using [dir()](https://www.programiz.com/python-programming/methods/built-in/dir) function. Some of the commonly used methods and attributes of match objects are:

### **match.group()**

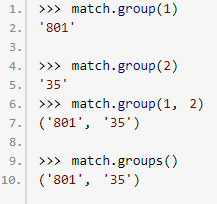
The group() method returns the part of the string where there is a match.

### **Example 6: Match object**



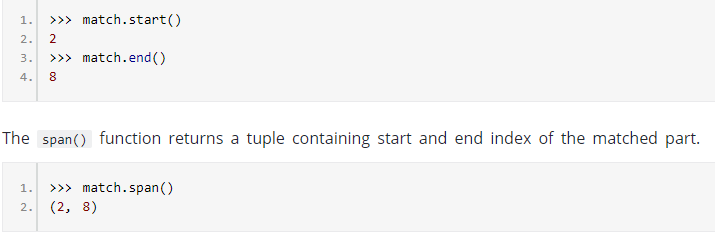
Here, match variable contains a match object.

Our pattern (\d{3}) (\d{2}) has two subgroups (\d{3}) and (\d{2}). You can get the part of the string of these parenthesized subgroups. Here's how:



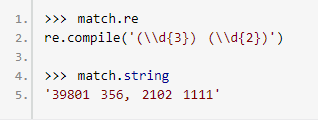
### **match.start(), match.end() and match.span()**

The start() function returns the index of the start of the matched substring. Similarly, end()returns the end index of the matched substring.



### **match.re and match.string**

The re attribute of a matched object returns a regular expression object. Similarly, string attribute returns the passed string.



We have covered all commonly used methods defined in the re module. If you want to learn more, visit [Python 3 re module](https://docs.python.org/3/library/re.html).

### **Using r prefix before RegEx**

When r or R prefix is used before a regular expression, it means raw string. For example, '\n' is a new line whereas r'\n' means two characters: a backslash \ followed by n. Backlash \ is used to escape various characters including all metacharacters. However, using r prefix makes \ treat as a normal character.

### **Example 7: Raw string using r prefix**

