**Lambda Expressions**

we will learn about the lambda expression,

**lambda expression**

One of Pythons most useful (and for beginners, confusing) tools is the lambda expression. lambda expressions allow us to create "anonymous" functions. This basically means we can quickly make ad-hoc functions without needing to properly define a function using def.

Function objects returned by running lambda expressions work exactly the same as those created and assigned by defs. There is key difference that makes lambda useful in specialized roles:

**lambda's body is a single expression, not a block of statements.**

* The lambda's body is similar to what we would put in a def body's return statement. We simply type the result as an expression instead of explicitly returning it. Because it is limited to an expression, a lambda is less general that a def. We can only squeeze design, to limit program nesting. lambda is designed for coding simple functions, and def handles the larger tasks.

**def** square(num):

result **=** num**\*\***2

**return** result

​

square(10)

​

**def** square(num):

**return** num**\*\***2

​

**def** square(num):**return** num**\*\***2

​

sq **=** **lambda** num : num**\*\***2

sq(10)

​

sq **=** **lambda** num : num**\*\***2

sq(10)

Out[1]:

100

Lets slowly break down a lambda expression by deconstructing a function:

We could simplify it:

We could actually even write this all on one line.

This is the form a function that a lambda expression intends to replicate. A lambda expression can then be written as:

**lambda** num: num **\*\*** 2

Out[1]:

<function \_\_main\_\_.<lambda>(num)>

*# You wouldn't usually assign a name to a lambda expression, this is just for demonstration!*

square **=** **lambda** num: num **\*\***2

square(2)

Out[26]:

4

So why would use this? Many function calls need a function passed in, such as map and filter. Often you only need to use the function you are passing in once, so instead of formally defining it, you just use the lambda expression. Let's repeat some of the examples from above with a lambda expression

list(map(**lambda** num: num **\*\*** 2, my\_nums))

Out[29]:

[1, 4, 9, 16, 25]

list(filter(**lambda** n: n **%** 2 **==** 0,nums))

Out[30]:

[0, 2, 4, 6, 8, 10]

Type *Markdown* and LaTeX: *α*2

\*\* Lambda expression for grabbing the first character of a string: \*\*

f**=lambda** s: s[0]

​

print(f('hello'))

h

\*\* Lambda expression for reversing a string: \*\*

f **=lambda** s: s[::**-**1]

print(f('hello'))

​

olleh

You can even pass in multiple arguments into a lambda expression. Again, keep in mind that not every function can be translated into a lambda expression.

a **=** **lambda** x,y : x **+** y

a(2,3)

Out[5]:

5

using lambda expressions often with certain non-built-in libraries, for example the pandas library for data analysis works very well with lambda expressions.

Python provides three functions that are widely used to implement functional programming style solutions in combination with collection container types. These functions are what are known as higher-order functions that take both a collection and a function that will be applied in various ways to that collection.

# Map

is another higher order function available in Python. Map applies the supplied function to all items in the iterable(s) passed to it. It returns a new iterable of the results generated by the applied function. It is the functional equivalent of a for loop applied to an iterable where the results of each iteration round the for loop are gathered up. The map function is very widely used within the functional programming world and it is certainly worth becoming familiar with it. The function signature of map is map(function, iterable, ...) Note that the second argument to the map function is anything that implements the iterable protocol. The function passed into the map function is applied to each item in the iterable passed as the second argument. The result returned from the function is then gathered up into the iterable object returned from map. The following example applies a function that adds one to a number, to a list of integers:

map(x**+**10 ,[1,2,3]) **=**[11,12,13]

map() is a built-in Python function that takes in two or more arguments: a function and one or more iterables, in the form:

map(function, iterable, ...)

map() returns an iterator - that is, map() returns a special object that yields one result at a time as needed..

When we went over list comprehensions we created a small expression to convert Celsius to Fahrenheit. Let's do the same here but use map:

## map function

The **map** function allows you to "map" a function to an iterable object. That is to say you can quickly call the same function to every item in an iterable, such as a list. For example:

data **=** [1, 3, 5, 2, 7, 4, 10]

print('data:', data)

*# Apply the lambda function to each element in the list*

*# using the map function*

​

​

​

​

print(list(map(**lambda** i:i**+**1 , data)))

​

[2,4,6,3,8,5,11]

​

​

data: [1, 3, 5, 2, 7, 4, 10]

[2, 4, 6, 3, 8, 5, 11]

[2, 4, 6, 3, 8, 5, 11]

data **=** [1, 3, 5, 2, 7, 4, 10]

print('data:', data)

*# Apply the lambda function to each element in the list*

*# using the map function*

​

​

print(list(map(**lambda** i:i**+**1,data)))

data: [1, 3, 5, 2, 7, 4, 10]

[2, 4, 6, 3, 8, 5, 11]

**def** add\_one(i):

**return** i **+** 1

*# Apply the add\_one function to each element in the*

*# list using the map function*

d2 **=** list(map(add\_one, data))

print('d2:', d2)

d2: [2, 4, 6, 3, 8, 5, 11]

As with the filter() function, the function to be applied can either be defined in line as a lambda or it can be named function as in add\_one(). Either can be used, the advantage of the add\_one() named function is that it makes the intent of the function explicit; however, it does pollute the namespace of functions defined.

**def** square(num):

**return** num**\*\***2

my\_nums **=** [1,2,3,4,5]

tuple(square,my\_nums)

Out[4]:

<map at 0x6228ed0>

*# To get the results, either iterate through map()*

*# or just cast to a list*

print(list(map(square,my\_nums)))

[1, 4, 9, 16, 25]

*# The functions can also be more complex*

​

**def** splicer(mystring):

**if** len(mystring) **%** 2 **==** 0:

**return** 'even'

**else**:

**return** mystring[0]

mynames **=** ['John','Cindy','Sarah','Kelly','Mike']

*#if functions are more complex then go for using the functions*

list(map(splicer,mynames))

Out[12]:

['even', 'C', 'S', 'K', 'even']

**def** fahrenheit(celsius):

**return** (9**/**5)**\***celsius **+** 32

temps **=** [0, 22.5, 40, 100]

Now let's see map() in action:

F\_temps **=** map(fahrenheit, temps)

​

*#Show*

list(F\_temps)

Out[2]:

[32.0, 72.5, 104.0, 212.0]

In the example above, map() applies the fahrenheit function to every item in temps. However, we don't have to define our functions beforehand; we can use a lambda expression instead:

list(map(**lambda** x: (9**/**5)**\***x **+** 32, temps))

Out[3]:

[32.0, 72.5, 104.0, 212.0]

list1**=**[1,2,4,5,8]

list2**=**list(map(**lambda** x:x**\***2, list1))

print(list1)

print(list2)

[1, 2, 4, 5, 8]

[2, 4, 8, 10, 16]

list1**=**[1,2,4,5,8]

ans**=**{x **for** x **in** (map(**lambda** x: x**\***2,list1))}

type(ans)

ans

Out[3]:

{2, 4, 8, 10, 16}

*#lambda does not return any value nor takes more than one exp and dict needs 2 values*

​

list1**=**[1,2,4,5,8]

ans **=** {list1[x]:y **for** x,y **in** enumerate((map(**lambda** y: y**\*\***2,list1)))} *#1,4,16,25,64*

ans

Out[4]:

{1: 1, 2: 4, 4: 16, 5: 25, 8: 64}

Great! We got the same result! Using map with lambda expressions is much more common since the entire purpose of map() is to save effort on having to create manual for loops.

### map() with multiple iterables

map() can accept more than one iterable. The iterables should be the same length - in the event that they are not, map() will stop as soon as the shortest iterable is exhausted.

For instance, if our function is trying to add two values **x** and **y**, we can pass a list of **x** values and another list of **y** values to map(). The function (or lambda) will be fed the 0th index from each list, and then the 1st index, and so on until the n-th index is reached.

Let's see this in action with two and then three lists:

a **=** [1,2,3,4]

b **=** [5,6,7,8]

c **=** [9,10,11,12]

​

list(map(**lambda** x,y:x**+**y,a,b)) *#here explicitly we need to specify 2 lists so that each one can be added*

*#this is in contrast with reduce*

Out[4]:

[6, 8, 10, 12]

*# Now all three lists*

list(map(**lambda** x,y,z:x**+**y**+**z,a,b,c))

Out[5]:

[15, 18, 21, 24]

We can see in the example above that the parameter **x** gets its values from the list **a**, while **y** gets its values from **b** and **z** from list **c**. Go ahead and play with your own example to make sure you fully understand mapping to more than one iterable.

Great job! You should now have a basic understanding of the map() function.

# filter

The filter() function is a higher order function that takes a function to be used to filter out elements from a collection. The result of the filter() function is a new iterable containing only those elements selected by the test function. That is, the function passed into filter() is used to test all the elements in the collection that is also passed into filter. Those where the test filter returns True are included in the list of values returned. The result returned is a new iterable consisting of all elements of this list that satisfy the given test function. Note that the order of the elements is preserved. The syntax of the filter() function is filter(function, iterable) Note that the second argument to the filter function is anything that implements the iterable protocol which includes all Lists, Tuples, Sets and dictionaries or and many other types etc. The function passed in as the first argument is the test function; it can be a lambda (a function defined in line) or the name of an existing function. The result returned will be an iterable that can be used to create an appropriate collection.

The function filter(function, list) offers a convenient way to filter out all the elements of an iterable, for which the function returns True.

The function filter(function,list) needs a function as its first argument. The function needs to return a Boolean value (either True or False). This function will be applied to every element of the iterable. Only if the function returns True will the element of the iterable be included in the result.

Like map(), filter() returns an iterator - that is, filter yields one result at a time as needed.

filter function The filter function returns an iterator yielding those items of iterable for which function(item) is true. Meaning you need to filter by a function that returns either True or False. T hen passing that into filter (along with your iterable) and you will get back only the results that would return True when passed to the function.

*# Here are some examples of using filter with a simple list of integers:*

data **=** [1, 3, 5, 2, 7, 4, 10]

print('data:', data)

​

​

print(list(filter(**lambda** i: **False**,data))) *# the filter function will take the true and false values returned by the lambda*

*# return value is none #and not any thing from the lambda*

print(list(filter(**lambda** i: **True**,data)))

​

data: [1, 3, 5, 2, 7, 4, 10]

[]

[1, 3, 5, 2, 7, 4, 10]

lst**=**[1,3,4,5,6,3,2]

print(list((map(**lambda** i: i**+**10,lst))))

​

**for** x **in** filter(**lambda** i: i**%**2**==**0,lst):

print(x)

*# [false,false,true,false,true,false,true]*

​

​

​

​

[11, 13, 14, 15, 16, 13, 12]

4

6

2

*# Filter for even numbers using a lambda function*

data **=** [1, 3, 5, 2, 7, 4, 10]

d1 **=** list(filter(**lambda** i: i **%** 2 **==** 0, data))

print('d1:', d1)

​

*# [false,false,false,true,false,true,true]*

*# [2,4,10]*

​

​

​

d1 **=** list(map(**lambda** i: i **%** 2 **==** 0, data))

print('d1:', d1)

d1: [2, 4, 10]

d1: [False, False, False, True, False, True, True]

data **=** [1, 3, 5, 2, 7, 4, 10]

d1 **=** list(filter(**lambda** i: i **%** 2 **==** 0, data))

print('d1:', d1)

d1: [2, 4, 10]

**def** is\_even(i):

**return** i **%** 2 **==** 0

*# Filter for even numbers using a named function*

d2 **=** list(filter(is\_even, data))

print('d2:', d2)

d2: [2, 4, 10]

One difference between the two examples is that it is more obvious what the role is of the test function in the second example as it is explicitly named (i.e. is\_even()), that is the function is testing the integer to see whether it is even or not. The in-line lambda function does exactly the same, but it is necessary to understand the test function itself to work out what it is doing. It is also worth pointing out that defining a named function such as is\_even() may actually pollute the namespace of the module as there is now a function that others might decide to use even though the original designer of this code never expected anyone else to use the is\_even() function. This is why lambda functions are often used with filter() (and indeed map() and reduce()).

**def** check\_even(num):

**return** num **%** 2 **==** 0

nums **=** [0,1,2,3,4,5,6,7,8,9,10]

filter(check\_even,nums)

list(filter(check\_even,nums))

*#First let's make a function*

**def** even\_check(num):

**if** num**%**2 **==**0:

**return** **True**

Now let's filter a list of numbers. Note: putting the function into filter without any parentheses might feel strange, but keep in mind that functions are objects as well.

lst **=**range(20)

​

list(filter(even\_check,lst))

Out[2]:

[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]

filter() is more commonly used with lambda functions, because we usually use filter for a quick job where we don't want to write an entire function. Let's repeat the example above using a lambda expression:

list(filter(**lambda** x: x**%**2**==**0,lst))

Out[3]:

[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]

**def** fun(x):

**if** x**%**2**==**0:

**return** **True**

**else**:

**return** **False**

list1**=**[1,28,3,21,23,22,11]

list2**=**list(filter(fun, list1))

print(list1)

print(list2)

[1, 28, 3, 21, 23, 22, 11]

[28, 22]

list1**=**[1,28,3,21,23,22,11]

**def** func(x):

**if** x**%** 2 **==** 0:

**return** **True**

**else**:

**return** **False**

list2 **=** list(filter(**lambda** x:x**%**2 **==**0,list1))

list2

Out[5]:

[28, 22]

list1**=**[1,28,3,21,23,22,11]

list2**=**list(filter(**lambda** x:x**%**2**==**0, list1))

print(list1)

print(list2)

[1, 28, 3, 21, 23, 22, 11]

[28, 22]

*#The filter value will be in terms of True or false and filter function will take the values of true value*

*#hence when used with map it will only show true and false , so only filter will convert the true false values to its*

*#corresponding values*

list1**=**[1,28,3,21,23,22,11]

list2**=**list(map(**lambda** x:x**%**2**==**0, list1))

print(list1)

print(list2)

[1, 28, 3, 21, 23, 22, 11]

[False, True, False, False, False, True, False]

Great! You should now have a solid understanding of filter() and how to apply it to your code!

**reduce()**

The reduce() function is the last higher order function that can be used with collections of data that we will look at. The reduce() function applies a function to an iterable and combines the result returned for each element together into a single result. This function was part of the core Python 2 language but was not included into the core of Python 3. This is partly because Guido van Rossum believed (probably correctly) that the applicability of reduce is quite limited, but where it is useful it is very useful. Although it has to be said that some developers try and shoe horn reduce() into situations that just make the implementation very hard to understand—remember always aim to keep it simple. To use reduce() in Python 3 you need to import it from the functoolsmodule. One point that is sometimes misunderstood with reduce() is that the function passed into reduce takes two parameters, which are the previous result and the next value in the sequence; it then returns the result of applying some operation to these parameters. The signature of the functools.reduce function is: functools.reduce(function, iterable[, initializer])

Many times students have difficulty understanding reduce() so pay careful attention to this lecture. The function reduce(function, sequence) continually applies the function to the sequence. It then returns a single value.

If seq = [ s1, s2, s3, ... , sn ], calling reduce(function, sequence) works like this:

* At first the first two elements of seq will be applied to function, i.e. func(s1,s2)
* The list on which reduce() works looks now like this: [ function(s1, s2), s3, ... , sn ]
* In the next step the function will be applied on the previous result and the third element of the list, i.e. function(function(s1, s2),s3)
* The list looks like this now: [ function(function(s1, s2),s3), ... , sn ]
* It continues like this until just one element is left and return this element as the result of reduce()

Let's see an example:

*# import functools*

**from** functools **import** reduce

​

lst **=**[47,11,42,13]

reduce(**lambda** x,y: x**+**y,lst)

Out[4]:

113

*# reduce function*

*# reduces sequence of elements into single value*

*#reduce(function,sequence) , it belongs to functools modules*

**from** functools **import** **\***

lst **=**[1,2,3,4,5]

​

reduce(**lambda** x,y : x **\***y,lst)

Out[7]:

120

Lets look at a diagram to get a better understanding of what is going on here:

**from** IPython.display **import** Image

Image('http://www.python-course.eu/images/reduce\_diagram.png')

Out[2]:

Diagram

Description automatically generated

Note how we keep reducing the sequence until a single final value is obtained. Lets see another example:

*#Find the maximum of a sequence (This already exists as max())*

lst **=** [2,6,1,8,9]

​

max\_find **=** **lambda** a,b: a **if** (a **>** b) **else** b

*#Find max*

reduce(max\_find,lst)

Out[7]:

9

**from** functools **import** reduce

list1**=**[1,28,3,21,23,22,11]

sum**=**reduce(**lambda** x,y:x**+**y, list1)

print("sum is ", sum)

*#Same code if we replace with map then it will give error because for reducing we need 2 values and*

*#2 values need to be supplied , but in map we have only one list object*

*# from functools import reduce*

*# list1=[1,28,3,21,23,22,11]*

*# sum=list(map(lambda x,y:x+y, list1))*

*# print("sum is ", sum)*

**---------------------------------------------------------------------------**

**TypeError** Traceback (most recent call last)

**<ipython-input-1-1e5fbd8e2887>** in <module>

1 **from** functools **import** reduce

2 list1**=[1,28,3,21,23,22,11]**

**----> 3** sum**=**list**(**map**(lambda** x**,**y**:**x**+**y**,** list1**))**

4 print**("sum is ",** sum**)**

**TypeError**: <lambda>() missing 1 required positional argument: 'y'

list1**=**[1,2,4,5,8,3]

*#i want sum of values at even locations/ indexes of a list*

*# 1+4+8=13*

​

**from** functools **import** reduce

list1**=**[1,2,4,5,8,3]

sumall **=** reduce(**lambda** x,y:x**+**y, list1[0::2])

print(sumall)

Hopefully you can see how useful reduce can be in various situations. Keep it in mind as you think about your code projects!

# zip

zip() makes an iterator that aggregates elements from each of the iterables.

Returns an iterator of tuples, where the i-th tuple contains the i-th element from each of the argument sequences or iterables. The iterator stops when the shortest input iterable is exhausted. With a single iterable argument, it returns an iterator of 1-tuples. With no arguments, it returns an empty iterator.

zip() is equivalent to:

def zip(\*iterables):

# zip('ABCD', 'xy') --> Ax By

sentinel = object()

iterators = [iter(it) for it in iterables]

while iterators:

result = []

for it in iterators:

elem = next(it, sentinel)

if elem is sentinel:

return

result.append(elem)

yield tuple(result)

zip() should only be used with unequal length inputs when you don’t care about trailing, unmatched values from the longer iterables.

Let's see it in action in some examples:

## Examples

x **=** [1,2,3]

y **=** [4,5,6]

​

​

print(zip(x,y))

*# Zip the lists together*

list(zip(x,y))

<zip object at 0x06748120>

Out[2]:

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

s **=** 'abcdef'

​

​

print(iter(s))

print(**\***iter(s)) *#what needs to be iterated*

print(**\***[iter(s)]) *#converted to string from iterator object , so that we can multiple with 3*

print(zip(**\***[iter(s)]))

print(list(zip(**\***[iter(s)]))) *#iterate through list and get indiviual values*

print(list(zip(**\***[iter(s)]**\***2))) *#for zipping 2 values are considered like a and b , c and d*

print(list(zip(**\***[iter(s)]**\***3))) *# for zipping 3 values are considered , since we do not have 2 objects , one string object only*

*#it needs to zip*

​

​

<str\_iterator object at 0x05D67150>

a b c d e f

<str\_iterator object at 0x05D67150>

<zip object at 0x05AEEB48>

[('a',), ('b',), ('c',), ('d',), ('e',), ('f',)]

[('a', 'b'), ('c', 'd'), ('e', 'f')]

[('a', 'b', 'c'), ('d', 'e', 'f')]

s **=** 'abcdef'

print(list(s))

print(zip(**\***[iter(s)]**\***3))

['a', 'b', 'c', 'd', 'e', 'f']

<zip object at 0x00EADF08>

Note how tuples are returned. What if one iterable is longer than the other?

x **=** [1,2,3]

y **=** [4,5,6,7,8]

​

*# Zip the lists together*

dict(zip(x,y))

Out[2]:

{1: 4, 2: 5, 3: 6}

Note how the zip is defined by the shortest iterable length. Its generally advised not to zip unequal length iterables unless your very sure you only need partial tuple pairings.

What happens if we try to zip together dictionaries?

d1 **=** {'a':1,'b':2}

d2 **=** {'c':4,'d':5}

​

t**=**list(zip(d1,d2))

x,y**=**t[0]

t[1]

Out[3]:

[('a', 'c'), ('b', 'd')]

This makes sense because simply iterating through the dictionaries will result in just the keys. We would have to call methods to mix keys and values:

list(zip(d2,d1.values()))

Out[4]:

[('c', 1), ('d', 2)]

Great! Finally lets use zip() to switch the keys and values of the two dictionaries:

**def** switcharoo(d1,d2):

dout **=** {}

**for** d1key,d2val **in** zip(d1,d2.values()):

dout[d1key] **=** d2val

**return** dout

switcharoo(d1,d2)

Out[6]:

{'a': 4, 'b': 5}

Great! You can use zip to save a lot of typing in many situations! You should now have a good understanding of zip() and some possible use cases.

# enumerate()

In this lecture we will learn about an extremely useful built-in function: enumerate(). Enumerate allows you to keep a count as you iterate through an object. It does this by returning a tuple in the form (count,element). The function itself is equivalent to:

def enumerate(sequence, start=0):

n = start

for elem in sequence:

yield n, elem

n += 1

## Example

lst **=** ['a','b','c']

​

**for** number,item **in** enumerate(lst):

print(number)

print(item)

0

a

1

b

2

c

enumerate() becomes particularly useful when you have a case where you need to have some sort of tracker. For example:

**for** count,item **in** enumerate(lst):

**if** count **>=** 2:

**break**

**else**:

print(item)

a

b

enumerate() takes an optional "start" argument to override the default value of zero:

months **=** ['March','April','May','June']

​

**for** count,item **in** enumerate(months,start**=**3):

print(count,item)

​

*# or can be done in this way also*

list(enumerate(months,start**=**3))

3 March

4 April

5 May

6 June

Out[2]:

[(3, 'March'), (4, 'April'), (5, 'May'), (6, 'June')]

Great! You should now have a good understanding of enumerate and its potential use cases.

# all() and any()

all() and any() are built-in functions in Python that allow us to conveniently check for boolean matching in an iterable. all() will return True if all elements in an iterable are True. It is the same as this function code:

def all(iterable):

for element in iterable:

if not element:

return False

return True

any() will return True if any of the elements in the iterable are True. It is equivalent to the following function code:

def any(iterable):

for element in iterable:

if element:

return True

return False

Let's see a few examples of these functions. They should be fairly straightforward:

lst **=** [**True**,**True**,**False**,**True**]

all(lst)

Out[2]:

False

Returns False because not all elements are True.

any(lst)

Out[3]:

True

Returns True because at least one of the elements in the list is True

There you have it, you should have an understanding of how to use any() and all() in your code.

# Iterators and Generators

**def** gen\_num():

**for** item **in** range(10):

**return** item

​

'''for i in gen\_num() will be error as int is not iterable'''

​

**for** i **in** range(10):

print(gen\_num())

**def** gen\_num1():

**for** item **in** range(10):

**yield** item

​

**for** i **in** gen\_num1():

print(i)

0

0

0

0

0

0

0

0

0

0

lst **=**[1,2,3,4]

**for** item **in** lst:

**yield** item

​

In this section of the course we will be learning the difference between iteration and generation in Python and how to construct our own Generators with the yield statement. Generators allow us to generate as we go along, instead of holding everything in memory.

We've touched on this topic in the past when discussing certain built-in Python functions like **range()**, **map()** and **filter()**.

Let's explore a little deeper. We've learned how to create functions with def and the return statement. Generator functions allow us to write a function that can send back a value and then later resume to pick up where it left off. This type of function is a generator in Python, allowing us to generate a sequence of values over time. The main difference in syntax will be the use of a yield statement.

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.

In many cases it is not appropriate (or possible) to obtain all the data to be processed up front (for performance reasons, for memory reasons etc.). Instead lazily creating the data to be iterated over based on some underlying dataset, may be more appropriate. Generators are a special function that can be used to generate a sequence of values to be iterated over on demand (that is when the values are needed) rather than produced up front. The only thing that makes a generator a generator function is the use of the yield keyword (which was introduced in Python 2.3). The yield keyword can only be used inside a function or a method. Upon its execution the function is suspended, and the value of the yield statement is returned as the current cycle value. If this is used with a for loop, then the loop runs once for this value. Execution of the generator function is then resumed after the loop has cycled once and the next cycle value is obtained. The generator function will keep supplying values until it returns (which means that an infinite sequence of values can be generated).

# Defining a Generator Function

A very simple example of a generator function is given below. This function is called the gen\_numbers() function:

**def** gen\_numbers():

**yield** 1 *#return 1*

**yield** 2

**yield** 3

**for** i **in** gen\_numbers():

print(i)

​

1

2

3

This is a generator function as it has at least one yield statement (in fact it has three). Each time the gen\_numbers() function is called within a for statement it will return one of the values associated with a yield statement; in this case the value 1, then the value 2 and finally the value 3 before it returns (terminates).

It is common for the body of a generator to have some form of loop itself. This loop is typically used to generate the values that will be yielded. However, as is shown above that is not necessary and here a yield statement is repeated three times. Note that gen\_numbers() is a function but it is a special function as it returns a generator object. This is a generator function returns a generator object which wraps up the generation of the values required but this is hidden from the developer.

When Do the Yield Statements Execute? It is interesting to consider what happens within the generator function; it is actually suspended each time a yield statement supplies a value and is only resumed when the next request for a value is received. This can be seen by adding some additional print statements to the gen\_numbers() function:

**def** gen\_numbers2():

print('Start')

**yield** 1

print('Continue')

**yield** 2

print('Final')

**yield** 3

print('End')

**for** i **in** gen\_numbers2():

print("Enter")

print(i)

​

*# Thus the generator executes the yield statements on an as needed basis and not*

*# all at once.*

Start

Enter

1

Continue

Enter

2

Final

Enter

3

End

*# Generator function for the cube of numbers (power of 3)*

**def** gencubes(n):

**for** num **in** range(n):

**yield** num**\*\***3

​

​

*# gencubes(10)*

**for** x **in** gencubes(10):

print(x)

0

1

8

27

64

125

216

343

512

729

Now since we have a generator function we don't have to keep track of every single cube we created.

Generators are best for calculating large sets of results (particularly in calculations that involve loops themselves) in cases where we don’t want to allocate the memory for all of the results at the same time.

Let's create another example generator which calculates [fibonacci](https://en.wikipedia.org/wiki/Fibonacci_number" \t "_blank) numbers:

**def** genfibon(n):

"""

Generate a fibonnaci sequence up to n

"""

a **=** 1

b **=** 1

**for** i **in** range(n):

**yield** a

a,b **=** b,a**+**b

**for** num **in** genfibon(10):

print(num)

1

1

2

3

5

8

13

21

34

55

What if this was a normal function, what would it look like?

**def** fibon(n):

a **=** 1

b **=** 1

output **=** []

**for** i **in** range(n):

output.append(a)

a,b **=** b,a**+**b

**return** output

fibon(10)

Out[6]:

[1, 1, 2, 3, 5, 8, 13, 21, 34, 55]

Notice that if we call some huge value of n (like 100000) the second function will have to keep track of every single result, when in our case we actually only care about the previous result to generate the next one!

*# An Even Number Generator*

*# We could have used a generator to produce a set of even numbers up to a specific limit,*

*# as we did earlier with the Evens class, but without the need to create a class (and*

*# implement the two special methods \_\_iter\_\_() and \_\_next\_\_()). For example:*

**def** evens\_up\_to(limit):

value **=** 0

**while** value **<=** limit:

**yield** value

value **+=** 2

**for** i **in** evens\_up\_to(6):

print(i, end**=**', ')

​

​

0, 2, 4, 6,

​

This illustrates the potential benefit of a generator over an iterator; the evens\_up\_to() function is a lot simpler and concise then the Evens iterable class.

# Nesting Generator Functions

You can even nest generator functions as each call to the generator function is encapsulated in its own generator object which captures all the state information needed by that generator invocation. For example:

**for** i **in** evens\_up\_to(4):

print('i:', i)

**for** j **in** evens\_up\_to(6):

print('j:', j, end**=**', ')

print('')

i: 0

j: 0, j: 2, j: 4, j: 6,

i: 2

j: 0, j: 2, j: 4, j: 6,

i: 4

j: 0, j: 2, j: 4, j: 6,

As you can see from this the loop variable i is bound to the values produced by the first call to evens\_up\_to() (which produces a sequence up to 4) while the j loop variable is bound to the values produced by the second call to evens\_up\_to() (which produces a sequence of values up to 6).

# Using Generators Outside a for Loop

You do not need a for loop to work with a generator function; the generator object actually returned by the generator function supports the next() function. This function takes a generator object (returned from the generator function) and returns the next value in sequence.

Subsequent calls to next(evens) return no value; if required the generator can throw an error/exception.

​

evens **=** evens\_up\_to(4)

print(next(evens), end**=**', ')

print(next(evens), end**=**', ')

print(next(evens))

0, 2, 4

# Iterators

An iterator is an object that will return a sequence of values. Iterators may be finite in length or infinite (although many container-oriented iterators provide a fixed set of values). The iterator protocol specifies the **next**() method. This method is expected to return the next item in the sequence to return or to raise the StopIteration exception. This is used to indicate that the iterator has finished supplying values

# The Iteration Related Methods

To summarise then we have

• **iter**() from the Iterable protocol which is used to return the iterator object,

• **next**() from the Iterator protocol which is used to obtain the next value in a sequence of values.

Any data type can be both an Iterable and an Iterator; but that is not required. An Iterable could return a different object that will be used to implement the iterator or it can return itself as the iterator—it’s the designers choice.

The Itertools Module The itertools module provides a number of useful functions that return iterators constructed in various ways. It can be used to provide an iterator over a selection of values from a data type that is iterable; it can be used to combine iterables together etc

## next() and iter() built-in functions

A key to fully understanding generators is the next() function and the iter() function.

The next() function allows us to access the next element in a sequence. Lets check it out:

**def** simple\_gen():

**for** x **in** range(3):

**yield** x

*# Assign simple\_gen*

g **=** simple\_gen()

​

*# return value of generator function is iteratable object*

g

​

**for** i **in** simple\_gen():

print(i)

​

​

Out[29]:

<generator object mygen at 0x08527DF0>

print(next(g))

0

print(next(g))

1

print(next(g))

2

print(next(g))

**---------------------------------------------------------------------------**

**StopIteration** Traceback (most recent call last)

**<ipython-input-12-1dfb29d6357e>** in <module>**()**

**----> 1** print**(**next**(**g**))**

**StopIteration**:

After yielding all the values next() caused a StopIteration error. What this error informs us of is that all the values have been yielded.

You might be wondering that why don’t we get this error while using a for loop? A for loop automatically catches this error and stops calling next().

Let's go ahead and check out how to use iter(). You remember that strings are iterables:

s **=** 'hello'

​

*#Iterate over string*

**for** let **in** s:

print(let)

h

e

l

l

o

But that doesn't mean the string itself is an iterator! We can check this with the next() function:

next(s)

**---------------------------------------------------------------------------**

**TypeError** Traceback (most recent call last)

**<ipython-input-14-61c30b5fe1d5>** in <module>**()**

**----> 1** next**(**s**)**

**TypeError**: 'str' object is not an iterator

Interesting, this means that a string object supports iteration, but we can not directly iterate over it as we could with a generator function. The iter() function allows us to do just that!

s\_iter **=** iter(s)

next(s\_iter)

Out[16]:

'h'

next(s\_iter)

Out[17]:

'e'

know how to convert objects that are iterable into iterators themselves!

using the yield keyword at a function will cause the function to become a generator. This change can save you a lot of memory for large use cases. For more information on generators check out:

[Stack Overflow Answer](http://stackoverflow.com/questions/1756096/understanding-generators-in-python)

[Another StackOverflow Answer](http://stackoverflow.com/questions/231767/what-does-the-yield-keyword-do-in-python)

*# Generators*

*# Generators are functions that return a sequence of values .it will be written like other functions but will be using yield st*

​

**def** mygen(x,y):

**while**(x **<=**y):

**yield** x

x**+=**1

g **=** mygen(5,10)

g

Out[3]:

<generator object mygen at 0x08527DF0>

**for** i **in** g:

print(i,end **=**' ')

5 6 7 8 9 10

*# if we wanted to retrive element by element from generator object, we can use next() function as*

**def** mygen():

**yield** 'A'

**yield** 'b'

**yield** 'c'

​

g**=** mygen() *# calling generator function and get generator object g*

​

*# display all the characters in the generator*

print(next(g))

print(next(g))

print(next(g))

​

*#print(next(g)) will give error*