Easily Explained

ADVANCED PYTHON TIPS

A SIMPLE BOOK ON ADVANCED PYTHON CONCEPTS

BY RAHUL AGARWAL

RAHUL AGARWAL

Advanced Python Tips

Explained Simply

First edition

This book was professionally typeset on Reedsy.
Find out more at reedsy.com

Contents

About me

I am Rahul Agarwal(MLWhiz), a data scientist consultant, and big data engineer based in Bangalore, where I am currently working with **WalmartLabs**.

Previously, I have worked at startups like **Fractal** and **MyCity-Way** and conglomerates like **Citi**. I started my blog mlwhiz.com with a purpose to augment my own understanding of new things while helping others learn about them. I also write for publications on Medium like **Towards Data Science** and **HackerNoon**

As Feynman said: **"I couldn't do it. I couldn't reduce it to the freshman level. That means we don't really understand it"**

Personally I am tool agnostic. I like learning new tools and constantly work to add up new skills as I face new problems that cannot be accomplished with my current set of techniques. But the tools that get most of my work done currently are Python, Hadoop, and Spark.

I also really like working with data-intensive problems and am constantly in search of new ideas to work on.

Introduction

Learning a language is easy. Whenever I start with a new language, I focus on a few things in the below order, and it is a breeze to get started with writing code in any language.

- Operators and Data Types: +,-,int,float,str
- Conditional statements: if,else,case,switch
- Loops: For, while
- Data structures: List, Array, Dict, Hashmaps
- Define Function

However, learning to write a language and writing a language in an optimized way are two different things.

Every Language has some ingredients which make it unique.

Yet, ***a new programmer to any language will always do some forced overfitting.*** A Java programmer, new to python, for example, might write this code to add numbers in a list.

```
x=[1,2,3,4,5]
sum_x = 0
for i in range(len(x)):
    sum_x+=x[i]
```

While a Python programmer will naturally do this:

```
sum_x = sum(x)
```

In this book, I will explain some simple constructs provided by Python, some essential tips, and some use cases I come up with regularly in my Data Science work. Most of the book is of a practical nature and you will find it beaming with examples.

This book is about efficient and readable code.

This book is distributed as free to read/pay as you want. If you like it, I would appreciate it if you could buy the paid version here.

1

Minimize for loop usage in Python

There are many ways to write a for loop in python.

A beginner may get confused on what to use.

Let me explain this with a simple example statement.

Suppose you want to take the ***sum of squares in a list.***

This is a valid problem we all face in machine learning whenever we want to calculate the distance between two points in n dimension.

You can do this using loops easily.

In fact, I will show you **three ways to do the same task which I have seen people use and let you choose for yourself which you find the best.**

```
x = [1,3,5,7,9]
sum_squared = 0
for i in range(len(x)):
    sum_squared+=x[i]**2
```

Whenever I see the above code in a python codebase, I understand that the person has come from C or Java background.

A **slightly more pythonic way** of doing the same thing is:

```
x = [1,3,5,7,9]
sum_squared = 0
for y in x:
    sum_squared+=y**2
```

Better.

I didn't index the list. And my code is more readable.

But still, the pythonic way to do it is in one line.

```
x = [1,3,5,7,9]
sum_squared = sum([y**2 for y in x])
```

This approach is called List Comprehension, and this may very well be one of the reasons that I love Python.

You can also use ***if*** in a list comprehension.

Let's say we wanted a list of squared numbers for even numbers only.

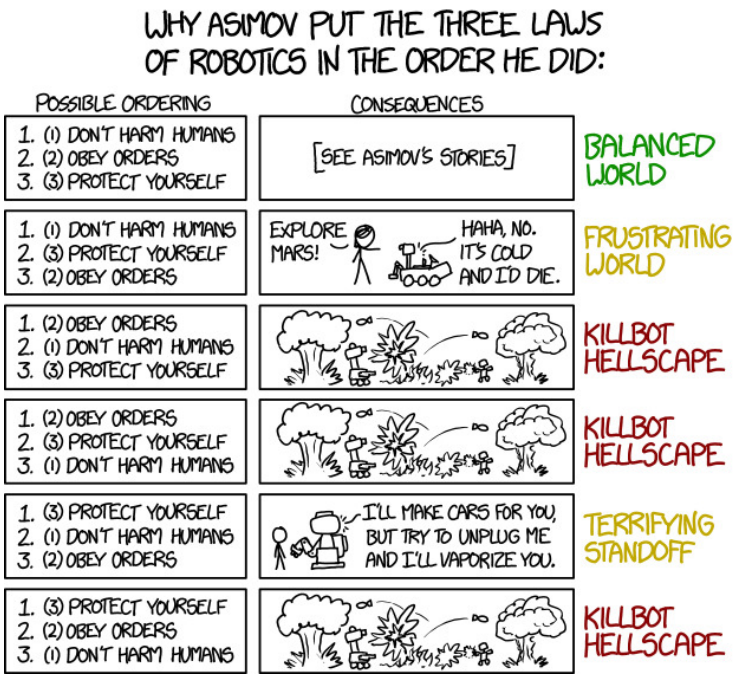
```
x = [1,2,3,4,5,6,7,8,9]
even_squared = [y**2 for y in x if y%2==0]
-------------------------------------------
[4,16,36,64]
```

What about if-else?

What if we wanted to have the number squared for even and cubed for odd?

```
x = [1,2,3,4,5,6,7,8,9]
squared_cubed = [y**2 if y%2==0 else y**3 for y in x]
-------------------------------------------
[1, 4, 27, 16, 125, 36, 343, 64, 729]
```

Great!!!



So basically follow specific **guidelines:** Whenever you feel like writing a forstatement, you should ask yourself the following questions,

- Can it be done without a ***for*** loop? Most Pythonic
- Can it be done using **list comprehension**? If yes, use it.
- Can I do it without indexing arrays? if not, think about using ***enumerate***

What is enumerate?

Sometimes we need both the index in an array as well as the value in an array.

In such cases, I prefer to use **enumerate** rather than indexing the list.

```
L = ['blue', 'yellow', 'orange']
for i, val in enumerate(L):
    print("index is %d and value is %s" % (i, val))
-------------------------------------------------------------
index is 0 and value is blue
index is 1 and value is yellow
index is 2 and value is orange
```

The rule is:

> Never index a list, if you can do without it.

Try Using Dictionary Comprehension

Also try using **dictionary comprehension**, which is a relatively new addition in Python. The syntax is pretty similar to List comprehension.

Let me explain using an example. I want to get a dictionary with (key: squared value) for every value in x.

```
x = [1,2,3,4,5,6,7,8,9]
{k:k**2 for k in x}
-----------------------------------------------------------
{1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64,
9: 81}
```

What if I want a dict only for even values?

```
x = [1,2,3,4,5,6,7,8,9]
{k:k**2 for k in x if x%2==0}
------------------------------------------------------------
{2: 4, 4: 16, 6: 36, 8: 64}
```

What if we want squared value for even key and cubed number for the odd key?

```
x = [1,2,3,4,5,6,7,8,9]
{k:k**2 if k%2==0 else k**3 for k in x}
------------------------------------------------------------
{1: 1, 2: 4, 3: 27, 4: 16, 5: 125, 6: 36, 7: 343, 8:
64, 9: 729}
```

Conclusion

To conclude, I will say that while it might seem easy to transfer the knowledge you acquired from other languages to Python, you won't be able to appreciate the beauty of Python if you keep doing that.

Python is much more powerful when we use its ways and decidedly much more fun.

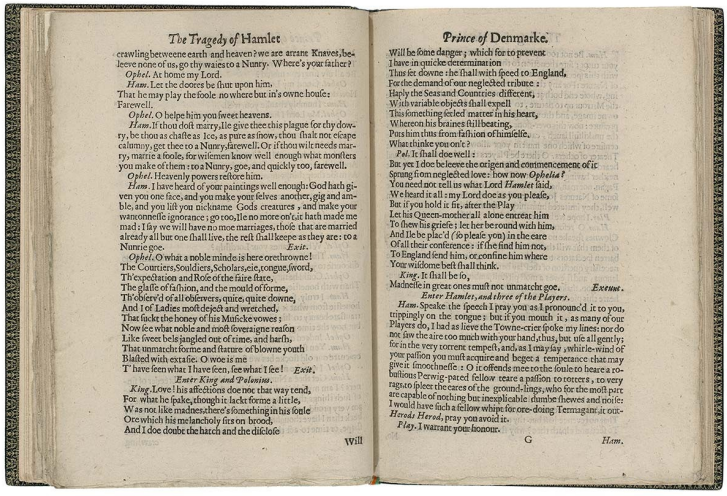
So, use List Comprehensions and Dict comprehensions when you need a for loop. Use enumerate if you need array index.

> Avoid for loops like plague

Your code will be much more readable and maintainable in the long run.

2

Python defaultdict and Counter



The Tragedy of Hamlet

crawling betweene earth and heaven? we are arrant Knaues, beleeue none of vs, go thy waies to a Nunry. Where's your father?

Ophel. At home my Lord.

Ham. Let the doores be shut vpon him,
That he may play the foole no where but in's owne house:
Farewell.

Ophel. O helpe him you sweet heavens.

Ham. If thou doſt marry, Ile giue thee this plague for thy dowry, be thou as chaſte as Ice, as pure as snow, thou ſhalt not eſcape calumny, get thee to a Nunry, farewell. Or if thou wilt needs marry, marrie a foole, for wiſemen know well enough what monſters you make of them: to a Nunry, goe, and quickly too, farewell.

Ophel. Heauenly powers reſtore him.

Ham. I haue heard of your paintings well enough: God hath giuen you one face, and you make your ſelues another, gig and amble, and you lift you nickname Gods creatures, and make your wantonneſſe ignorance: go too, Ile no more on't, it hath made me mad: I ſay we will haue no moe marriages, thoſe that are married already all but one ſhall liue, the reſt ſhall keepe as they are: to a Nunrie goe. *Exit.*

Ophel. O what a noble minde is here orethrowne!
The Courtiers, Souldiers, Scholars, eie, tongue, ſword,
Th'expectation and Roſe of the faire ſtate,
The glaſſe of fashion, and the mould of forme,
Th'obſerv'd of all obſeruers, quite, quite downe,
And I of Ladies moſt deiect and wretched,
That ſucke the honey of his Muſicke vowes:
Now ſee what noble and moſt ſoueraigne reaſon
Like ſweet bels iangled out of time, and harſh,
That vnmatcht forme and ſtature of blowne youth
Blaſted with extaſie. O woe is me
T'haue ſeen what I haue ſeen, ſee what I ſee! *Exit.*

Enter King and Polonius.

King. Loue! his affections doe not that way tend,
Nor what he ſpake, though it lack'd forme a little,
Was not like madneſs: there's ſomething in his ſoule
Ore which his melancholy ſits on brood,
And I doe doubt the hatch and the diſcloſe

Will

Prince of Denmarke.

Will be ſome danger; which for to preuent
I haue in quicke determination
Thus ſet downe: he ſhall with ſpeed to England,
For the demand of our neglected tribute:
Haply the Seas and Countries different,
With variable obiects ſhall expell
This ſomething ſetled matter in his heart,
Whereon his braines ſtill beating,
Puts him thus from fashion of himſelfe.
What thinke you on't?

Pol. It ſhall doe well:
But yet I doe beleeue the origen and commencement of it
Sprung from neglected loue: how now *Ophelia?*
You need not tell us what Lord *Hamlet* ſaid,
We heard it all: my Lord doe as you pleaſe,
But if you hold it fit, after the Play
Let his Queen-mother all alone entreat him
To ſhew his griefe: let her be round with him,
And I be plac'd (ſo pleaſe you) in the eare
Of all their conference: if ſhe find him not,
To England ſend him, or confine him where
Your wiſedome beſt ſhall think.

King. It ſhall be ſo,
Madneſſe in great ones muſt not vnmatcht goe. *Exeunt.*

Enter Hamlet, and three of the Players.

Ham. Speake the ſpeech I pray you as I pronounc'd it to you, trippingly on the tongue: but if you mouth it, as many of our Players do, I had as lieue the Towne-crier ſpoke my lines: nor do not ſaw the aire too much with your hand, thus, but vſe all gently; for in the very torrent, tempeſt, and as I may ſay, whirle-wind of your paſſion you muſt acquire and beget a temperance that may giue it ſmoothneſſe: O it offends mee to the ſoule to heare a robuſtious Periwig-pated fellow teare a paſſion to totters, to very rags, to ſpleet the eares of the groundlings, who for the moſt part are capable of nothing but inexplicable dumbe ſhewes and noiſe: I would haue ſuch a fellow whipt for ore-doing Termagant; it out-

Herod Herod, pray you auoid it.

Play. I warrant your honour.

G *Ham.*

Let's say I need to count the number of word occurrences in a piece of text.

Maybe for a book like Hamlet. How could I do that?

Python always provides us with multiple ways to do the same thing. But only one way I find elegant.

This is a ***Naive Python implementation*** using the dict object.

```
text = "I need to count the number of word
occurrences in a piece of text. How could I do that?
Python provides us with multiple ways to do the same
thing. But only one way I find beautiful."

word_count_dict = {}

for w in text.split(" "):
    if w in word_count_dict:
        word_count_dict[w]+=1
    else:
        word_count_dict[w]=1
```

We could use ***defaultdict*** to reduce the number of lines in the code.

```
from collections import defaultdict
word_count_dict = defaultdict(int)
for w in text.split(" "):
    word_count_dict[w]+=1
```

We could also have used ***Counter*** to do this, which according to me is the most preferable method for this problem.

```
from collections import Counter
word_count_dict = Counter()
for w in text.split(" "):
```

```
    word_count_dict[w]+=1
```

If we use Counter, we can also get the most common words using a simple function.

```
word_count_dict.most_common(10)
-------------------------------------------------------------
[('I', 3), ('to', 2), ('the', 2)]
```

Other use cases of Counter:

```
# Count Characters
Counter('abccccccddddd')
-------------------------------------------------------------
Counter({'a': 1, 'b': 1, 'c': 6, 'd': 5})
# Count List elements
Counter([1,2,3,4,5,1,2])
-------------------------------------------------------------
Counter({1: 2, 2: 2, 3: 1, 4: 1, 5: 1})
```

* * *

So, why ever use defaultdict?

Notice that in *Counter*, the value is always an integer.

What if we wanted to parse through a list of tuples containing colors and fruits. And wanted to create a dictionary of key and list of values.

The main functionality provided by a defaultdict is that it

defaults a key to empty/zero if it is not found in the defaultdict.

```
s = [('color', 'blue'), ('color', 'orange'),
('color', 'yellow'), ('fruit', 'banana'), ('fruit',
'orange'),('fruit','banana')]
d = defaultdict(list)
for k, v in s:
     d[k].append(v)
print(d)
----------------------------------------------------------------
defaultdict(<class 'list'>, {'color': ['blue',
'orange', 'yellow'], 'fruit': ['banana', 'orange',
'banana']})
```

banana comes two times in fruit, we could use set

```
d = defaultdict(set)
for k, v in s:
     d[k].add(v)
print(d)
----------------------------------------------------------------
defaultdict(<class 'set'>, {'color': {'yellow',
'blue', 'orange'}, 'fruit': {'banana', 'orange'}})
```

* * *

Conclusion

To conclude, I will say that ***there is always a beautiful way to do anything in Python***. Search for it before you write code. Going to StackOverflow is okay. I go there a lot of times when I get stuck. Always Remember:

Creating a function for what already is provided is not pythonic.

3

*args, **kwargs, decorators for Data Scientists

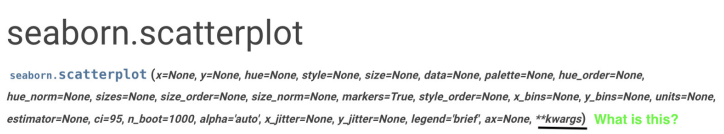
Python has a lot of constructs that are reasonably easy to learn and use in our code.

Then there are some constructs which always confuse us when we encounter them in our code.

Then are some that even seasoned programmers are not able to understand. *args, **kwargs and decorators are some constructs that fall into this category.

I guess a lot of my data science friends have faced them too.

Most of the seaborn functions use *args and **kwargs in some way or other.



Or what about decorators?

Every time you see a warning like some function will be deprecated in the next version. The sklearn package uses decorators for that. You can see the ***@deprecated*** in the source code. That is a decorator function.

```
from ..utils import deprecated

def zero_one_loss(y_true, y_pred, normalize=True):
    # actual implementation
    pass

@deprecated("Function 'zero_one' was renamed to 'zero_one_loss' "
            "in version 0.13 and will be removed in release 0.15. "
            "Default behavior is changed from 'normalize=False' to "
            "'normalize=True'")
def zero_one(y_true, y_pred, normalize=False):
    return zero_one_loss(y_true, y_pred, normalize)
```

* * *

What are *args?

In simple terms, ***you can use *args to give an arbitrary number of inputs to your function.***

A simple example:

Let us say we have to create a function that adds two numbers. We can do this easily in python.

```
def adder(x,y):
    return x+y
```

What if we want to create a function to add three variables?

```
def adder(x,y,z):
    return x+y+z
```

What if we want the same function to add an unknown number of variables?

Please note that we can use ****args*** or ****argv*** or ****anyOtherName*** to do this. It is the * that matters.

```
def adder(*args):
    result = 0
    for arg in args:
        result+=arg
    return result
```

What ****args*** does is that it takes all your passed arguments and provides a variable length argument list to the function which you can use as you want.

Now you can use the same function as follows:

```
adder(1,2)
adder(1,2,3)
adder(1,2,5,7,8,9,100)
```

and so on.

Now, have you ever thought how the print function in python

could take so many arguments? *args

* * *

What are **kwargs?

In simple terms, ***you can use **kwargs to give an arbitrary number of Keyworded inputs to your function*** and access them using a dictionary.

A simple example:

Let's say you want to create a print function that can take a name and age as input and print that.

```
def myprint(name,age):
    print(f'{name} is {age} years old')
```

Simple. Let us now say you want the same function to take two names and two ages.

```
def myprint(name1,age1,name2,age2):
    print(f'{name1} is {age1} years old')
```

```
    print(f'{name2} is {age2} years old')
```

You guessed right my next question is: ***What if I don't know how many arguments I am going to need?***

Can I use *args? Guess not since name and age order is essential. We don't want to write "28 is Michael years old".

Come **kwargs in the picture.

```
def myprint(**kwargs):
    for k,v in kwargs.items():
        print(f'{k} is {v} years old')
```

You can call this function using:

```
myprint(Sansa=20,Tyrion=40,Arya=17)
Output:-----------------------------------
Sansa is 20 years old
Tyrion is 40 years old
Arya is 17 years old
```

Remember we never defined Sansa or Arya or Tyrion as our methods arguments.

That is a pretty powerful concept. And many programmers utilize this pretty cleverly when they write wrapper libraries.

For example, ***seaborn.scatterplot*** function wraps the ***plt.scatter*** function from Matplotlib.

Essentially, using *args and **kwargs we can provide all the

arguments that ***plt.scatter*** can take to ***seaborn.Scatterplot*** as well.

This can save a lot of coding effort and also makes the code future proof. If at any time in the future ***plt.scatter*** starts accepting any new arguments the ***seaborn.Scatterplot*** function will still work.

* * *

What are Decorators?

In simple terms: ***Decorators are functions that wrap another function thus modifying its behavior.***

A simple example:

Let us say we want to add custom functionality to some of our functions. The functionality is that whenever the function gets called the "***function name*** begins" is printed and whenever the function ends the "***function name*** ends" and time taken by the function is printed.

Let us assume our function is:

```
def somefunc(a,b):
    output = a+b
    return output
```

We can add some print lines to all our functions to achieve this.

```
import time
def somefunc(a,b):
    print("somefunc begins")
    start_time = time.time()
    output = a+b
    print("somefunc ends in ",time.time()-start_time,
    "secs")
    return output
out = somefunc(4,5)
OUTPUT:
----------------------------------------
somefunc begins
somefunc ends in  9.5367431640625e-07 secs
```

But, Can we do better?

This is where decorators excel. We can use decorators to wrap any function.

```
from functools import wraps
def timer(func):
    @wraps(func)
    def wrapper(a,b):
        print(f"{func.__name__!r} begins")
        start_time = time.time()
        result = func(a,b)
        print(f"{func.__name__!r} ends in
        {time.time()-start_time}  secs")
        return result
    return wrapper
```

This is how we can define any decorator. functools helps us create decorators using wraps. In essence, we do something before any function is called and do something after a function is called in the above decorator.

We can now use this timer decorator to decorate our function somefunc

```
@timer
def somefunc(a,b):
    output = a+b
    return output
```

Now calling this function, we get:

```
a = somefunc(4,5)
Output
----------------------------------------------
'somefunc' begins
'somefunc' ends in 2.86102294921875e-06  secs
```

Now we can append @timer to each of our function for which

we want to have the time printed. And we are done.

Really?

* * *

Connecting all the pieces

What if our function takes three arguments? Or many arguments?

This is where whatever we have learned till now connects. We use *args and **kwargs

We change our decorator function as:

```
from functools import wraps
def timer(func):
    @wraps(func)
    def wrapper(*args,**kwargs):
        print(f"{func.__name__!r} begins")
        start_time = time.time()
        result = func(*args,**kwargs)
        print(f"{func.__name__!r} ends in
        {time.time()-start_time}  secs")
        return result
    return wrapper
```

Now our function can take any number of arguments, and our decorator will still work.

Isn't Python Beautiful?

In my view, decorators could be pretty helpful. I provided only one use case of decorators, but there are several ways one can use them.

You can use a decorator to debug code by checking which arguments go in a function. Or a decorator could be used to count the number of times a particular function has been called. This could help with counting recursive calls.

* * *

Conclusion

In this post, I talked about some of the constructs you can find in python source code and how you can understand them.

It is not necessary that you end up using them in your code now. But I guess understanding how these things work helps mitigate some of the confusion and panic one faces whenever these constructs come up.

4

Use Itertools, Generators, and Generator Expressions

Python in many ways has made our life easier when it comes to programming.

With its many libraries and functionalities, sometimes we forget

to focus on some of the useful things it offers.

One of such functionalities are generators and generator expressions. I stalled learning about them for a long time but they are useful.

Have you ever encountered yield in Python code and didn't knew what it meant? or what does an iterator or a generator means and why we use it? Or have you used ImageDataGenerator while working with Keras and didn't understand what is going at the backend? Then this chapter is for you.

The Problem Statement:

Let us say that we need to run a for loop over 10 Million Prime

numbers.

I am using prime numbers in this case for understanding but it could be extended to a case where we have to process a lot of images or files in a database or big data.

How would you proceed with such a problem?

Simple. We can create a list and keep all the prime numbers there.

Really? ***Think of the memory such a list would occupy.***

It would be great if we had something that could just keep the last prime number we have checked and returns just the next prime number.

That is where iterators could help us.

* * *

The Iterator Solution

We create a class named primes and use it to generate primes.

```
def check_prime(number):
    for divisor in range(2, int(number ** 0.5) + 1):
        if number % divisor == 0:
```

```
            return False
    return True
class Primes:
    def __init__(self, max):
        # the maximum number of primes we want
        generated
        self.max = max
        # start with this number to check if it is a
        prime.
        self.number = 1
        # No of primes generated yet. We want to
        StopIteration when it reaches max
        self.primes_generated = 0
    def __iter__(self):
        return self
    def __next__(self):
        self.number += 1
        if self.primes_generated >= self.max:
            raise StopIteration
        elif check_prime(self.number):
            self.primes_generated+=1
            return self.number
        else:
            return self.__next__()
```

We can then use this as:

prime_generator = Primes(10000000)

```
for x in prime_generator:
    # Process Here
```

Here I have defined an iterator. This is how most of the functions like xrange or ImageGenerator work.

Every iterator needs to have:

1. an ___iter___ method that returns self, and
2. an ___next___ method that returns the next value.
3. a StopIteration exception that signifies the ending of the iterator.

Every iterator takes the above form and we can tweak the functions to our liking in this boilerplate code to do what we want to do.

See that we don't keep all the prime numbers in memory just the state of the iterator like

- what max prime number we have returned and
- how many primes we have returned already.

But it seems a little too much code. Can we do better?

* * *

The Generator Solution

Put simply Generators provide us ways to write iterators easily using the yield statement.

```
def Primes(max):
    number = 1
    generated = 0
    while generated < max:
        number += 1
        if check_prime(number):
            generated+=1
            yield number
```

we can use the function as:

```
prime_generator = Primes(10)
for x in prime_generator:
    # Process Here
```

It is so much simpler to read. But what is yield?

We can think of yield as a return statement only as it returns the value.

But when a yield happens the state of the function is also saved in the memory. So at every iteration in for loop the function variables like number, generated and max are stored somewhere in memory.

So what is happening is that the above function is taking care of all the boilerplate code for us by using the yield statement.

Much More pythonic.

* * *

Generator Expression Solution

While not explicitly better than the previous solution but we can also use Generator expression for the same task. But we might lose some functionality here. They work exactly like list comprehensions but they don't keep the whole list in memory.

primes = (i for i in range(1,100000000) if check_prime(i))

```
for x in primes:
    # do something
```

Functionality loss: We can generate primes till 10M. But we can't generate 10M primes. One can only do so much with generator expressions.

But generator expressions let us do some pretty cool things.

Let us say we wanted to have all Pythagorean Triplets lower than 1000.

How can we get it?

Using a generator, now we know how to use them.

```
def triplet(n): # Find all the Pythagorean triplets
between 1 and n
    for a in range(n):
        for b in range(a):
            for c in range(b):
                if a*a == b*b + c*c:
                    yield(a, b, c)
```

We can use this as:

```
triplet_generator = triplet(1000)
for x in triplet_generator:
    print(x)
-------------------------------------------------------------
(5, 4, 3)
(10, 8, 6)
(13, 12, 5)
(15, 12, 9)
.....
```

Or, we could also have used a generator expression here:

```
triplet_generator = ((a,b,c) for a in range(1000) for
b in range(a) for c in range(b) if a*a == b*b + c*c)
for x in triplet_generator:
    print(x)
-------------------------------------------------------------
(5, 4, 3)
(10, 8, 6)
(13, 12, 5)
(15, 12, 9)
.....
```

Isn't Python Beautiful?

I hate code blocks sometimes due to code breaks and copy paste issues. So you can see all the code in this kaggle kernel.

* * *

Conclusion

We must always try to reduce the memory footprint in Python. Iterators and generators provide us with a way to do that with Lazy evaluation.

How do we choose which one to use? What we can do with generator expressions we could have done with generators or iterators too.

There is no correct answer here. Whenever I face such a dilemma, I always think in the terms of functionality vs readability. Generally,

Functionality wise: Iterators>Generators>Generator Expressions.

Readability wise: Iterators<Generators<Generator Expressions.

It is not necessary that you end up using them in your code now. But I guess understanding how these things work helps mitigate some of the confusion and panic one faces whenever

these constructs come up.

5

How and Why to use f strings in Python3?

Python provides us with many styles of coding. And with time, Python has regularly come up with new coding standards and tools that adhere even more to the coding standards in the Zen of Python.

Beautiful is better than ugly.

And so this chapter is ***about using f strings in Python that was introduced in Python 3.6.***

* * *

3 Common Ways of Printing:

Let me explain this with a simple example. Suppose you have some variables, and you want to print them within a statement.

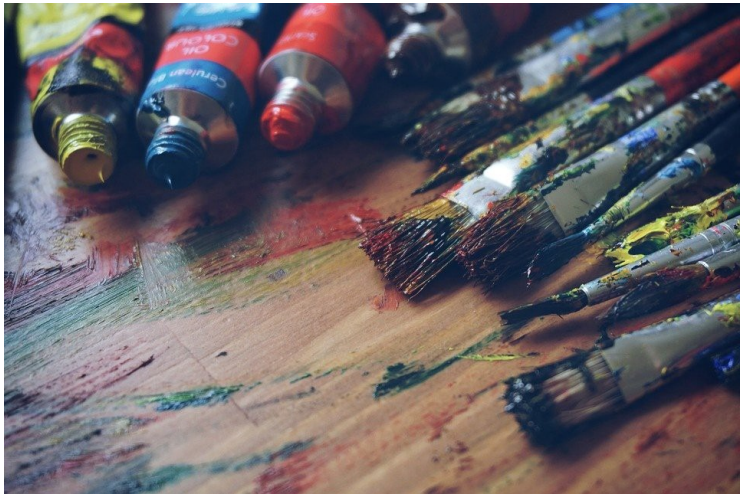
```
name = 'Andy'
age = 20
print(?)
-----------------------------------------------------------------
Output: I am Andy. I am 20 years old
```

You can do this in various ways:

a) Concatenate: A very naive way to do is to simply use + for concatenation within the print function. But that is clumsy. We would need to convert our numeric variables to string and keep care of the spaces while concatenating. And it doesn't look good

as the code readability suffers a little when we use it.

```
name = 'Andy'
age = 20
print("I am " + name + ". I am " + str(age) + " years
old")
-----------------------------------------------------------------
I am Andy. I am 20 years old
```



b) % Format: The second option is to use % formatting. But it also has its problems. For one, it is not readable. You would need to look at the first %s and try to find the corresponding variable in the list at the end. And imagine if you have a long list of variables that you may want to print.

```
print("I am %s. I am %s years old" % (name, age))
```

c) str.format(): Next comes the way that has been used in most Python 3 codes and has become the standard of printing in Python. Using str.format()

```
print("I am {}. I am {} years old".format(name, age))
```

Here we use {} to denote the placeholder of the object in the list. It still has the same problem of readability, but we can also use str.format :

```
print("I am {name}. I am {age} years old".format(name
= name, age = age))
```

If this seems a little too repetitive, we can use dictionaries too:

```
data = {'name':'Andy','age':20}
print("I am {name}. I am {age} years
old".format(**data))
```

* * *

The Fourth Way with f

Since Python 3.6, we have a new formatting option, which makes it even more trivial. We could simply use:

```
print(f"I am {name}. I am {age} years old")
```

We just append f at the start of the string and use {} to include our variable name, and we get the required results.

An added functionality that f string provides is that **we can put expressions** in the {} brackets. For Example:

```
num1 = 4
num2 = 5
print(f"The sum of {num1} and {num2} is {num1+num2}.")
-----------------------------------------------------------------
The sum of 4 and 5 is 9.
```

This is quite useful as you can use any sort of expression inside these brackets. ***The expression can contain dictionaries***

or functions. A simple example:

```
def totalFruits(apples,oranges):
    return apples+oranges

data = {'name':'Andy','age':20}

apples = 20
oranges = 30

print(f"{data['name']} has
{totalFruits(apples,oranges)} fruits")
-----------------------------------------------------------------
Andy has 50 fruits
```

Also, you can use ''' to use ***multiline strings***.

```
num1 = 4
num2 = 5
print(f'''The sum of
{num1} and
{num2} is
{num1+num2}.''')

-----------------------------------------------------------------
The sum of
4 and
5 is
9.
```

An everyday use case while formatting strings is to **format floats**. You can do that using f string as following

```
numFloat = 10.23456678
print(f'Printing Float with 2 decimals:
{numFloat:.2f}')
```

```
------------------------------------------------------------------
Printing Float with 2 decimals: 10.23
```

* * *

Conclusion

Until recently, I had been using Python 2 for all my work, and so was not able to check out this new feature. But now, as I am shifting to Python 3, f strings has become my go-to syntax to format strings. It is easy to write and read with the ability to incorporate arbitrary expressions as well. In a way, this new function adheres to at least 3 **PEP** concepts —

> *Beautiful is better than ugly, Simple is better than complex and Readability counts.*

Afterword

This is just a brief overview of a few of the functionalities in Python. I tried to make them as beginner-friendly as possible without going too much into terminologies. Let me know what you think of this short book on Twitter @mlwhiz.

I am still going to write more about new functionalities in Python. Follow me up at **Medium**. As always, I welcome feedback and constructive criticism and can be reached on Twitter @mlwhiz.

Also if you want to learn more about Python 3, I would like to call out an excellent course on Learn Intermediate level Python from the University of Michigan. Do check it out.

This book is distributed as free to read/pay as you want. If you like it, I would appreciate it if you could buy the paid version here.