5 Python features I wish I had known earlier

Python tricks beyond lambda, map, and filter

Eden Au Follow
Dec 25 · 4 min read ★

Photo by Kirill Sharkovski on Unsplash

Python is arguably the rising programming language of the decade and is proven to be a very powerful language. I have built so many applications using Python from interactive maps to blockchains. There are so many features in Python, and it is very difficult for beginners to grasp everything at first.

Even if you are a programmer switching from other languages such as C or MATLAB, coding in Python with a higher level of abstraction is definitely a different experience. I wish I had known some Python features earlier, and have highlighted five of the most important ones.

. . .

1. List comprehensions — compact codes

Many people would mention the **lambda**, **map**, and **filter** as Python 'tricks' that every beginner should learn. While I believe they are functions that we should be aware of, I find them not particularly useful most of the time as they **lack flexibility**.

`Lambda` is a method to compose a function in one line **for one-time use**. Performance suffers if the functions are called multiple times. On the other hand, `map` applies a function to all elements in a list, whereas `filter` gets a subset of elements in a set that meets a user-defined condition.

```
add_func = lambda z: z ** 2
is_odd = lambda z: z%2 == 1
multiply = lambda x,y: x*y

aList = list(range(10))
print(aList)
# [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

python-lambda.py hosted with ♡ by GitHub view raw

Photo by Anastase Maragos on Unsplash

List comprehension is a concise and flexible method to create lists from other lists with flexible expressions and conditions. It is constructed by a square bracket, with an expression or a function that is applied to every element in a list only if the element satisfies a certain condition. It can also be nested to handle nested lists, and is far more flexible than using map and filter.

```
# Syntax of list comprehension
[ expression(x) for x in aList if optional_condition(x) ]
```

```
print(list(map(add_func, aList)))
print([x ** 2 for x in aList])
# [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
# [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

print(list(filter(is_odd, aList)))
print([x for x in aList if x%2 == 1])
```

```
# [1, 3, 5, 7, 9]
# [1, 3, 5, 7, 9]
```

python-list-comprehension.py hosted with ♡ by GitHub view raw

. . .

2. List manipulation — circular lists

Python allows **negative indexing** where `aList[-1] == aList[len(aList)-1]`. Therefore, we can get the second last element in a list by calling `aList[-2]` and so on.

We can also **slice** lists using the syntax `aList[start:end:step]`, where the starting element is included but the ending element is not. Therefore, calling `aList[2:5]` gives `[2, 3, 4]`. We can also **reverse a list** simply by calling `aList[::-1]`, and I find this technique very elegant.

Photo by Martin Shreder on Unsplash

A list can be also **unpacked** into separate elements, or a mix of elements and a sub-list using an asterisk.

```
a, b, c, d = aList[0:4]
print(f'a = {a}, b = {b}, c = {c}, d = {d}')
# a = 0, b = 1, c = 2, d = 3

a, *b, c, d = aList
print(f'a = {a}, b = {b}, c = {c}, d = {d}')
# a = 0, b = [1, 2, 3, 4, 5, 6, 7], c = 8, d = 9
```

python-unpacking.py hosted with ♡ by GitHub view raw

. . .

3. Zipping and enumerate — for-loops

`Zip` function creates an **iterator** that aggregates elements from multiple lists. It allows **traversing lists in parallel** in a for-loop and **sorting in parallel**. It can be unzipped using an asterisk.

```
numList = [0, 1, 2]
engList = ['zero', 'one', 'two']
espList = ['cero', 'uno', 'dos']
```

```
print(list(zip(numList, engList, espList)))
# [(0, 'zero', 'cero'), (1, 'one', 'uno'), (2, 'two', 'dos')]

for num, eng, esp in zip(numList, engList, espList):
    print(f'{num} is {eng} in English and {esp} in Spanish.')
# 0 is zero in English and cero in Spanish.
# 1 is one in English and uno in Spanish.
# 2 is two in English and dos in Spanish.
```

python-zip-1.py hosted with ♡ by GitHub view raw

```
Eng = list(zip(engList, espList, numList))
Eng.sort() # sort by engList
a, b, c = zip(*Eng)

print(a)
print(b)
print(c)
# ('one', 'two', 'zero')
# ('uno', 'dos', 'cero')
# (1, 2, 0)
```

python-zip-2.py hosted with ♡ by GitHub view raw

Photo by Erol Ahmed on Unsplash

`Enumerate` might look a bit intimidating at first, but becomes very handy in many scenarios. It is an **automatic counter** that is often used in a for-loop, such that there is **no need to create and initialise a counter** variable anymore in a for-loop by `counter = 0` and `counter += 1`. Enumerate and zip are two of the most powerful tools when constructing a for-loop.

```
upperCase = ['A', 'B', 'C', 'D', 'E', 'F']
lowerCase = ['a', 'b', 'c', 'd', 'e', 'f']
for i, (upper, lower) in enumerate(zip(upperCase, lowerCase), 1):
    print(f'{i}: {upper} and {lower}.')
# 1: A and a.
# 2: B and b.
# 3: C and c.
# 4: D and d.
# 5: E and e.
# 6: F and f.
```

python-enumerate.py hosted with ♡ by GitHub view raw

. . .

4. Generator — memory efficiency

Generators are utilized when we intend to calculate a large set of results but would like to **avoid allocating the memory needed for all results at the same time**. In other words, they generate values **on the fly** and do not store previous values in memory, and thus we can only iterate over them once.

They are often used when reading large files or **generating an infinite sequence** using keyword `yield`. I often find it useful in most of my data science projects.

```python
def gen(n):    # an infinite sequence generator that generates integers >= n
    while True:
        yield n
        n += 1

G = gen(3)     # starts at 3
print(next(G)) # 3
print(next(G)) # 4
print(next(G)) # 5
print(next(G)) # 6
```

python-generator.py hosted with ♡ by GitHub view raw

. . .

5. Virtual environment — isolation

If you could only remember one thing in this article, then it should be the use of **virtual environments**.

Photo by Matthew Kwong on Unsplash

Python applications often use many different **packages** from various developers with complex **dependencies**. Different applications are developed using a specific library setting, where **results cannot be reproduced** using other library versions. There **does not exist a single installation** that satisfies the requirements of all applications.

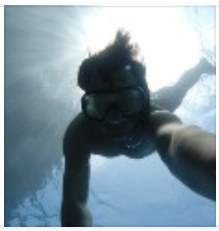
```
conda create -n venv pip python=3.7  # select python version
source activate venv
...
source deactivate
```

Therefore, it is vital to create separate self-contained virtual environments `venv` for each application, which can be done using `pip` or `conda`.

. . .

Related Articles

Thank you for reading. If you are interested in data science, the following articles might be useful:



4 NumPy Tricks Every Python Beginner should Learn

Tricks for writing readable codes

towardsdatascience.com

Visualizing Bike Mobility in London using Interactive Maps and Animations

Exploring data visualization tools in Python

towardsdatascience.com

Why Sample Variance is Divided by n-1

Explaining high school statistics that your teachers didn't teach

towardsdatascience.com

Originally published at edenau.github.io.

Thanks to Ludovic Benistant.

Python Data Programming Lambda Coding