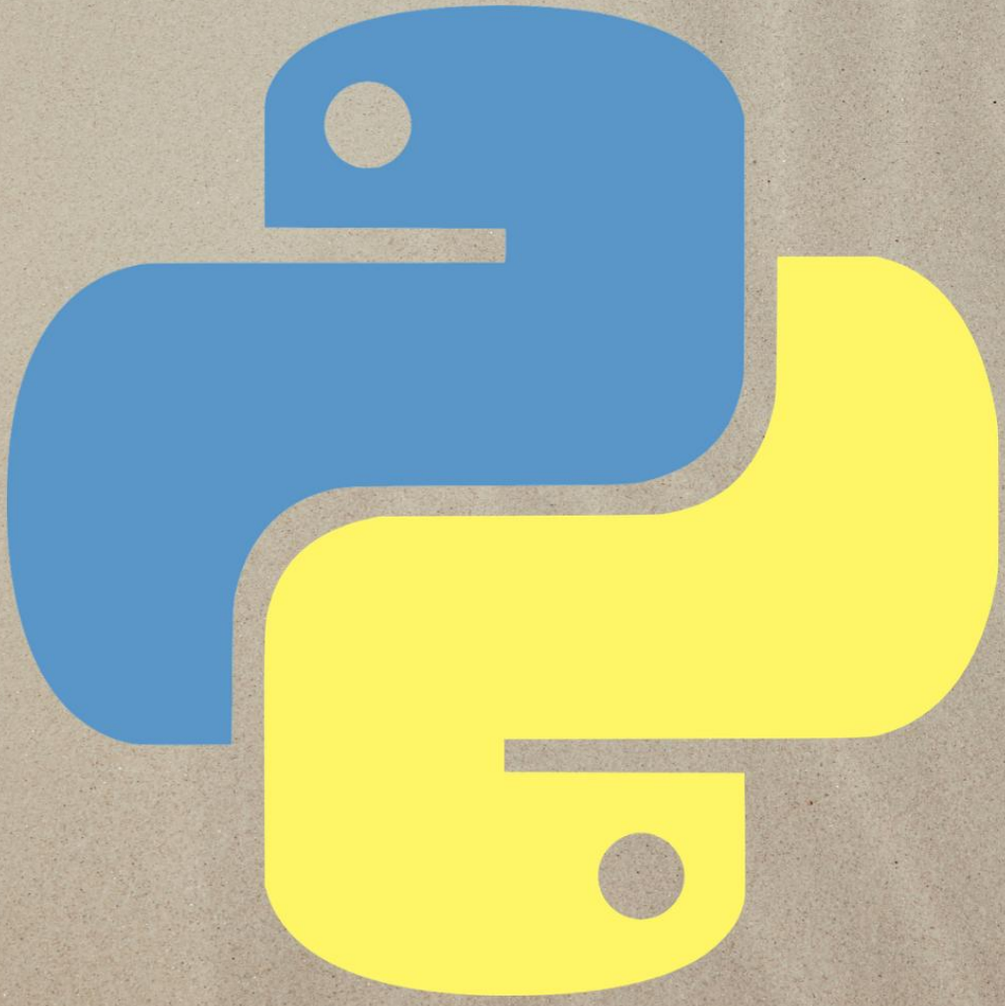
PYTHON TIPS & TRICKS

50 BASIC & INTERMEDIATE TIPS & TRICKS



Benjamin Bennett Alexander

PYTHON

TIPS & TRICKS

50 BASIC & INTERMEDIATE TIPS AND TRICKS

Benjamin Bennett Alexander

Feedback and Reviews

I welcome and appreciate your feedback and reviews. Feedback helps us independent writers reach more people. So please consider leaving feedback on the platform you got this book from. Send queries to: benjaminbennettalexander@gmail.com. Thanks.

Table of Contents

About this Book

This book is a collection of Python tips and tricks. I have put together 50 Python tips and tricks that you may find helpful if you are learning Python. The tips cover mainly basic and intermediate levels of Python.

In this book you will find tips on:
- How to print horizontally using the print function
- How to use list comprehensions to make code my concise
- How to update a dictionary using dictionary comprehensions
- How to merge dictionaries
- Swapping variables
- Merging PDF files
- Creating DataFrames using pandas
- Using sort method and sorted function to sort iterables
- Writing CSV files and so much more

To fully benefit, please try your best to write the code down and try running the code. It is important that you try yo understand how the code works. Modify and improve the code. Do not be scared to experiment.

By the end of it all, I hope that the tips and tricks help you level-up on your Python skills and knowledge.

Let's get started.

1. Printing Horizontally

When looping over an iterable, the print function prints each item in the new line. This is because the print function has a parameter called **end**. By default, the end parameter has an escape character (end ='\n'). To print horizontally, we need to remove the escape character and replace it with an empty string (end = " "). Here is the code to demonstrate that:

```
list1 = [1, 3, 6, 7]

for number in list1:
    print(number, end= " ")
```

```
Output:
1 3 6 7
```

The print function has another parameter called **sep**. We use *sep* to specify how the output is separated. Here is an example:

```
print('12','12','1990', sep='/')
```

```
Output:
12/12/1990
```

2. Merging Dictionaries

If you have two dictionaries that you want to combine, you can combine them using two easy methods. You can use the merge (|) operator or the (**) operator. Below, we have two dictionaries, a and b. We are going to use these two methods to combine the dictionaries. Here are the codes below:

Method 1

```
name1 = {"kelly": 23,
         "Derick": 14, "John": 7}
name2 = {"Ravi": 45, "Mpho": 67}

names = name1 | name2
print(names)
```

Output:

```
{'kelly': 23, 'Derick': 14, 'John': 7, 'Ravi': 45,
'Mpho': 67}
```

Method 2

```
name1 = {"kelly": 23,
         "Derick": 14, "John": 7}
name2 = {"Ravi": 45, "Mpho": 67}

names = {**name1, **name2}
print(names)
```

Output:

```
{'kelly': 23, 'Derick': 14, 'John': 7, 'Ravi': 45,
'Mpho': 67}
```

3. Calendar with Python

Did you know that you can get a calendar using Python?

Python has a built-in module called **calendar**. We can import this module to print out the calendar. We can do a lot of things with calendar.

Let's say we want to see April 2022; we will use the *month* class of the calendar module and pass the year and month as arguments. See below:

```
import calendar

month = calendar.month(2022, 4)
print(month)
```

Output:

```
  April 2022
Mo Tu We Th Fr Sa Su
                1  2  3
 4  5  6  7  8  9 10
11 12 13 14 15 16 17
18 19 20 21 22 23 24
25 26 27 28 29 30
```

4. Get Current Time

The below code demonstrates how you can get the current time using the *datetime()* module. The *strftime()* method to format time to the desired output. This code breaks down how you can use the *datetime* module with *strftime()* method to get a formatted string of time into hours, minutes, and seconds format.

```
from datetime import datetime

time_now = datetime.now().strftime('%H:%M:%S')
print(f'The time now is {time_now}')
Output:
The time now is 17:53:19
```

For more format codes, visit the website below:

https://www.w3schools.com/python/python_datetime.asp

5. Sort a List in Descending Order

The *sort()* method will sort a list in **ascending** order (default). For the sort method to work, the list should have the same data type of objects. You **cannot** sort a list mixed with different data types such as integers and strings. The *sort()* method has a parameter called reverse, to sort a list in descending order set reverse to **True**.

```
list1 = [2, 5, 6, 8, 1, 8, 9, 11]

list1.sort(reverse=True)
print(list1)
```

Output:

```
[11, 9, 8, 8, 6, 5, 2, 1]
```

Remember, *Sort()* is strictly a list method. You cannot use it to sort a **set**, a **tuple,** a **string** or a **dictionary**.

The **sort()** method does not return a new list, it sorts an existing list. If you try to create a new object using *sort()* method, it will return **None**. See below:

```
list1 = [2, 5, 6, 8, 1, 8, 9, 11]
list2 = list1.sort(reverse=True)
print(list2)
```

Output:

```
None
```

6. Swapping Variables

In Python, you can swap variables once they have been assigned to objects. Below, we initially assign 20 to x and 30 to y, but then we swap them; x becomes 30 and y becomes 20.

```
x, y = 20, 30
x, y = y, x
print('x is', x)
print('y is', y)
```

```
Output:
x is 30
y is 20
```

In Python we can also **XOR** ^ to swap variables. This is a three step method. Below, we are using ^ to swap the values of x and y.

```
x = 20
y = 30

# step one
x ^= y
# step two
y ^= x
# step two
x ^= y

print(f'x is: {x}')
print(f'y is: {y}')
```

Output:

```
x is: 30
y is: 20
```

7. Counting Item Occurrences

If you want to know how many times an item appears in an iterable, you can use the *Counter()* class from the *collection* module. The *Counter()* class will return a **dictionary** of how many times each item appears in an iterable. Let's say we want to know how many times the name Peter appears in the following list; here is how we can use the *Counter()* class of the collections module.

```
from collections import Counter

list1 = ['John','Kelly', 'Peter', 'Moses', 'Peter']

count_peter = Counter(list1).get("Peter")

print(f'The name Peter appears in the list '
      f'{count_peter} times.')
Output:
The name Peter appears in the list 2 times.
```

Another way you can do it is using a normal for loop. This is the naïve way. See below:

```
list1 = ['John','Kelly', 'Peter', 'Moses', 'Peter']

# Create a count variable
count = 0
for name in list1:
    if name == 'Peter':
        count +=1
print(f'The name Peter appears in the list'
      f' {count} times.')
Output:
The name Peter appears in the list 2 times.
```

8. Flatten a Nested List

I will share with you **three(3)** ways you can flatten a list. The first method is using a *for loop* and the second method is using *itertools* module, and the third method is using *list comprehension*.

```
list1 = [[1, 2, 3],[4, 5, 6]]
newlist = []
for list2 in list1:
    for j in list2:
         newlist.append(j)
print(newlist)
Output:
[1, 2, 3, 4, 5, 6]
```

Using itertools

```
import itertools

list1 = [[1, 2, 3],[[4, 5, 6]]]

flat_list(list(itertools.chain.from_iterable(list1)))
print(list1)
Output:
[1, 2, 3, 4, 5, 6]
```

Using list Comprehension

```
list1 = [[1, 2, 3],[4, 5, 6]]

flat_list= [i for j in list1 for i in j]
print(flat_list)
Output:
[1, 2, 3, 4, 5, 6]
```

9. Index of Biggest Number in List

Using the ***enumerate()***, ***max()*** and ***min()*** functions, we can find the index of the biggest and the smallest number in a list.

Find the index of the biggest number:

```
x = [12, 45, 67, 89, 34, 67, 13]

max_num = max(enumerate(x),
                  key = lambda x: x[1])
print('The index of the biggest num is',
      max_num[0])
```

```
Output:
The index of the biggest num is 3
```

Finding the index of the smallest number:

```
x = [12, 45, 67, 89, 34, 67, 13]

max_num = min(enumerate(x),
                  key = lambda x : x[1])
print('The index of the smallest num is',
      max_num[0])
```

```
Output:
The index of the smallest num is 0
```

10. Absolute Value of a Number

Let's say you have a negative number and you want to return the absolute value of the number; you can use the **abs()** function. The Python **abs()** function is used to return an absolute value of any number. Below, we demonstrate how we can return a list of absolute numbers from a list of numbers that have negative and positive numbers.

```
list1 = [-12, -45, -67, -89, -34, 67, -13]
print([abs(i) for i in list1])
```

Output:

```
[12, 45, 67, 89, 34, 67, 13]
```

You can also use abs on floating number and it will return the absolute value. See below:

```
a = -23.12
print(abs(a))
```

Output:

```
23.12
```

11. Adding a Thousand Separator

If you are working with big figures and you want to add a separator to make them more readable, you can use the *format()* function. See the example below:

```
a = [10989767, 9876780, 9908763]

new_list =['{:,}'.format(num) for num in a]
print(new_list)
Output:
['10,989,767', '9,876,780', '9,908,763']
```

Have you noticed that we are using list comprehension to add the separator? Cool thing , right?😊.

12. Startswith and Endswith Methods

The *startswith()* and *endswith()* are string methods that return True if a specified string starts or ends with a specified value.

Let's say you want to return all the names in a list that starts with 'a', here is how you would use *startwith()* to accomplish that:

Using startswith()

```
list1 = ['lemon','Orange',
         'apple', 'apricot']
new_list = [i for i in list1 if i.startswith('a')]
print(new_list)
Output:
['apple', 'apricot']
```

Using Endswith()

```
list1 = ['lemon','Orange',
         'apple', 'apricot']
new_list = [i for i in list1 if i.endswith('e')]
print(new_list)
```

Output:

```
['Orange', 'apple']
```

13. Nlargest and Nsmallest

Let's say you have a list of numbers and you want to return the 5 largest numbers from that list, or the 5 smallest numbers from a list. You cannot just use the *max()* and *min()* functions because they only return a single number. There is a Python built-in module that you can use and will make your life easier. It is called *heapq*.

Using Nlargest

```
import heapq

results = [12, 34, 67, 98, 90, 68, 55, 54, 64, 35]
max_list = heapq.nlargest(5, results)
print(max_list)
```

Output:

```
[98, 90, 68, 67, 64]
```

Using Nsmallest

```
Import headq

Results =[12, 34, 67, 98, 90, 68, 55, 54, 64, 35]
min_list = heapq.nsmallest(5, results)
print(min_list)
```

Output:

```
[12, 34, 35, 54, 55]
```

14. Checking for Anagram

You have got two strings, how would you go about checking if they are anagrams?

If you want to check if two strings are anagrams, you can use the *Counter()* class from the collections module. Below we are checking if *a* and *b* are anagrams.

```
from collections import Counter

a = 'lost'
b = 'stol'
print(Counter(a)==Counter(b))
```

Output:

```
True
```

You can also use the *sorted()* function to check if two strings are anagrams. See the code below:

```
a = 'lost'
b = 'stol'

if sorted(a)== sorted(b):
    print('Anagrams')
else:
    print("Not Anagrams")
```

Output:

```
Anagrams
```

15. Checking Internet Speed

Did you know that you can check internet speed with Python?

There is a module called speedtest that you can use to check the speed of your internet. You will have to install it using pip.

```
pip install speedtest-cli
```

Since the output of speedtest is in bits, we divide it by 8000000 to get the results in mb. Go on, test your internet speed using Python.

Checking download speed

```
import speedtest

d_speed = speedtest.Speedtest()
print(f'{d_speed.download()/8000000:.2f}mb')
Output:
213.78mb
```

Checking upload speed

```
import speedtest

up_speed = speedtest.Speedtest()
print(f'{up_speed.upload()/8000000:.2f}mb')
Output:
85.31mb
```

16. Python Reserved keywords

If you want to know the reserved keywords in Python, you can use the *help()* function. **Remember, you cannot use any of these words as variable names.** Your code will generate an error.

```
print(help('keywords'))
```

Output:

```
Here is a list of the Python keywords.  Enter any
keyword to get more help.

False                       break                     for
not

None                        class                     from
or

True                        continue                  global
pass

__peg_parser__          def                           if
raise

and                         del                       import
return

as                elif              in                try

assert                      else                      is
while

async                       except                    lambda
with

await                       finally                 nonlocal
yield

None
```

17. Properties and Methods

If you want to know the properties and methods of an object or module, use the dir() function. Below we are checking for properties and methods of a string object. You can also use *dir()* to check the properties and methods of modules. For instance, if you wanted to know the properties and methods of the collections module, you, can import the collections module and pass it as an argument to the function – print(dir(collections)).

```
a = 'I love Python'
print(dir(a))
```

Output:

```
['__add__',      '__class__',      '__contains__',
'__delattr__',   '__dir__',   '__doc__',   '__eq__',
'__format__',     '__ge__',     '__getattribute__',
'__getitem__',      '__getnewargs__',      '__gt__',
'__hash__',     '__init__',     '__init_subclass__',
'__iter__',    '__le__',    '__len__',    '__lt__',
'__mod__',    '__mul__',    '__ne__',    '__new__',
'__reduce__',      '__reduce_ex__',      '__repr__',
'__rmod__',       '__rmul__',       '__setattr__',
'__sizeof__',     '__str__',     '__subclasshook__',
'capitalize',   'casefold',   'center',   'count',
'encode',   'endswith',   'expandtabs',    'find',
'format',   'format_map',    'index',    'isalnum',
'isalpha',   'isascii',   'isdecimal',   'isdigit',
'isidentifier',       'islower',       'isnumeric',
'isprintable',  'isspace',  'istitle',  'isupper',
'join',  'ljust',  'lower',  'lstrip',  'maketrans',
'partition',    'removeprefix',    'removesuffix',
'replace',     'rfind',     'rindex',     'rjust',
'rpartition',    'rsplit',    'rstrip',    'split',
'splitlines',  'startswith',  'strip',  'swapcase',
'title', 'translate', 'upper', 'zfill']
```

18. Open a Website Using Python

Did you know that you can open a website using a Python script?

To open a website using Python, import the *webbrowser* module. This is a built-in module, so you don't have to install anything. Below, we are trying to open google.com using the *open()* method of the module. You can do this with any website if you know its URL.

```
import webbrowser

website = https://www.google.com/

# This opens a new tab in your browser
webbrowser.open_new_tab(website)

# This opens a new browser window
webbrowser.open_new(website)
```

19. Most Frequent in a String

Let's say you have a string and you want to find the most frequent element in the string; you can use the *max()* function. The max() function will count the items in the string and return the item that appears the most. You just have to pass the string *count* method to the key parameter. Let's use the string below to demonstrate this.

```
a = 'fecklessness'

most_frequent = max(a, key = a.count)
print(f'The most frequent letter is, '
      f'{most_frequent}')
```

```
Output
The most frequent letter is, s
```

Now, if there is more than one most frequent item in the string, the *max()* function will return the element that comes first alphabetically. For example, if the string above had 4 Fs and 4 Ss, the max function would return 'f' instead of 's' as the most frequent element because it comes first alphabetically.

We can also use the **Counter** class from the collections module. The *most_common*() method of this class will count how many times each element appears in the list and it will return all the elements and their counts, as a list of tuples. Below, we pass the parameter (1) because we want it to return the number one most common element of the list. If we pass (2), it will return the two most common elements.

```
import collections

a = 'fecklessness'
print(collections.Counter(a).most_common(1))
```

```
Output:
('s', 4)
```

20. Memory Size Check

Do you want to know how much memory an object is consuming in Python?

The *sys* module has a class that you can use for such a task. Here is a code to demonstrate this.

```
import sys

a = ['Love', 'Cold', 'Hot', 'Python']
b = {'Love', 'Cold', 'Hot', 'Python'}
print(f'The memory size of a list is '
      f'{sys.getsizeof(a)} ')
print(f'The memory size of a set is '
      f'{sys.getsizeof(b)} ')
```

Output:

```
The memory size of a list is 120
The memory size of a set is 216
```

As you can see, lists are more space efficient than sets.

21. Accessing Dictionary Keys

How do you access keys in a dictionary? Below are three different ways you can access the keys of a dictionary:

1. Using set comprehension

```
dict1 = {'name': 'Mary', 'age': 22,
         'student':True,'Country': 'UAE'}
print({i for i in dict1.keys()})
Output:
{'age', 'student', 'Country', 'name'}
```

2. Using the set() function

```
dict1 = {'name': 'Mary', 'age': 22,
         'student':True,'Country': 'UAE'}
print(set(dict1))
Output:
{'name', 'Country', 'student', 'age'}
```

3. Using the sorted() function

```
dict1 = {'name': 'Mary', 'age': 22,
         'student':True,'Country': 'UAE'}
print(sorted(dict1))
Output:
 ['Country', 'age', 'name', 'student']
```

22. Iterable or Not

Question:

How do you confirm if an object is iterable using code?

This is how you can use code to check if an item is *iterable* or not. We are using the *iter()* function.

```
a = ['i','love','working', 'with', 'Python']

try:
    iter_check = iter(a)
except TypeError:
    print('Object a not iterable')
else:
    print('Object a is iterable')

# Check the second object
b = 45.7

try:
    iter_check = iter(b)
except TypeError:
    print('Object b is not iterable')
else:
    print('Object b is iterable')
```

```
Output:
Object a is iterable
Object b is not iterable
```

23. Sorting a List of Tuples

You can sort a list of tuples using the *itemgetter()* function of the operator module. The itemgetter is passed as a key to the *sorted()* function. If you want to sort by the first name, you pass the index (0) to the itemgetter function. Below are different ways you can use the i*temgette*r to sort the list of tuples.

```
from operator import itemgetter

names = [('Ben','Jones'),('Peter','Parker'),
         ('Kelly','Isa')]

#Sort names by first name
sorted_names = sorted(names,key=itemgetter(0))
print('Sorted by first name',sorted_names)

# sort names by last name
sorted_names = sorted(names,key=itemgetter(1))
print('Sorted by last name',sorted_names)

# sort names by first name, then last name
sorted_names =
sorted(names,key=itemgetter(0,1))
print('Sorted by last name & first name',sorted_names)
```

Output:

```
Sorted by first name [('Ben', 'Jones'), ('Kelly', 'Isa'), ('Peter', 'Parker')]
Sorted by last name [('Kelly', 'Isa'), ('Ben', 'Jones'), ('Peter', 'Parker')]
Sorted by last name & first name [('Ben', 'Jones'), ('Kelly', 'Isa'),
    ('Peter', 'Parker')]
```

24. Sort List with Sorted & Lambda

The *sorted()* function is a high-order function because it takes another function as a parameter. Here we create a *lambda* function that is then passed as a parameter to the *sorted()* function. By using a negative index[-1], we are telling the *sorted()* function to sort the list in descending order..

```
list1 = ['Mary','Peter', 'Kelly']
a = lambda x: x[-1]
y = sorted(list1, key=a)
print(y)
Output:
['Peter', 'Mary', 'Kelly']
```

To sort the list in ascending order, we can just change the index to [:]. See below:

```
list1 = ['Mary','Peter', 'Kelly']
a = lambda x: x[:1]
y = sorted(list1, key=a)
print(y)
Output:
['Kelly', 'Mary', 'Peter']
```

Another easy way to sort the list in ascending order would be to use just the *sorted()* function. By default, it sorts an iterable in descending order. The key is optional, so we just leave it out.

```
list1 = ['Mary','Peter', 'Kelly']
list2 = sorted(list1)
print(list2)
Output
['Kelly', 'Mary', 'Peter']
```

25. Access News using Python

You can do a lot of things in Python, even read the news. You can access the news using the Python newspapers module. First, you have to install the module.

Pip install newspaper3k.

Below we access the title of the article. You can also access the article's text.

```
from newspaper import Article

news = Article("https://indianexpress.com/article/"
               "technology/gadgets/"
               "apple-discontinues-its-last-ipod-model-7910720/")

news.download()
news.parse()
print(news.title)
```

Output

```
End of an Era: Apple discontinues its last iPod model
```

26. A List of Tuples with Enumerate

Since enumerate counts (adds a counter) the items it loops over, you can use it to create a list of tuples. Below we create a list of tuples of days in a week ,from a list of days in a week. Enumerate has a parameter called **start**. Start is the index you want the count to begin. By default, the start is zero(0). Below, we have set start at one(1). You can put any number you want as start.

```
days = ["Sunday", "Monday", "Tuesday", "Wednesday",
        "Thursday", "Friday", "Saturday"]

days_tuples = list(enumerate(days, start=1))
print(days_tuples)
```

Output:

```
[(1, 'Sunday'), (2, 'Monday'), (3, 'Tuesday'), (4, 'Wednesday'), (5, 'Thursday'), (6, 'Friday'), (7, 'Saturday')]
```

Tuples properties

1. They are surrounded by round brackets ( ).
2. The are ordered
3. They are immutable
4. They can hold arbitray objects.

27. Windows Warning Sound

Did you know that if you run **print('\a')** your windows machine will make a warning sound? Try this code below:

```
import time

for i in range(5):
    for j in range(4):
        for k in range(3):
            time.sleep(0.3)
            print('\a')
        print('\a')
        time.sleep(1)
```

28. Print Colored Texts

Did you know that you can add color to your code using Python ansi escape codes: Below, I created a class of codes and I apply it to the code that I print out.

```
class Colors():
    Black = '\033[30m'
    Green = '\033[32m'
    Blue = '\033[34m'
    Magenta = '\033[35m'
    Red = '\033[31m'
    Cyan = '\033[36m'
    White = '\033[37m'
    Yellow = '\033[33m'


print(f'{Colors.Red} Warning: {Colors.Green} '
      f'Love Don\'t live here anymore')
```

Output:

```
Warning:  Love Don't live here anymore
```

29. Find Index Using Enumerate

The simplest way to access the index of items in an iterable is using the *enumerate()* function. The enumerate function returns the item and its index, by default.

Let's say we want to find the index of the letter 'n' in str1 below. Here is how we can use the enumerate function to achieve that.

```
str1 = 'string'

for i, j in enumerate(str1):
    if j =='n':
        print(f"The index of n is {i}")
Output
The index of 'n' is 4
```

30. A Class Using the Type Function

The *type()* function is usually used to check the type of an object. However, it can also be used to create classes dynamically in Python. Below I have created two classes, the first one using the class keyword and the second one using the *type()* function. You can see that both methods achieve the same results.

```
# Creating dynamic class using the class keyword
class Car:
    def __init__(self, name, color):
        self.name = name
        self.color = color

    def print_car(self):
        return f'The car is {self.name} ' \
               f'and its {self.color} in color'

car1 = Car('BMW', 'Green')
print(car1.print_car())

# Creating dynamic class using the type keyword
def cars_init(self, name ,color):
    self.name = name
    self.color = color
Cars = type("Car",(),
            {'__init__': cars_init,
            'print_car':lambda  self:
            f'The car is {self.name} '
            f'and its {self.color} in color'})

car1 = Cars("BMW", "Green")
print(car1.print_car())
```

Output:

```
The car is BMW and its Green in color
The car is BMW and its Green in color
```

31. Checking if a String is Empty

The simple way to check if a given string is empty is to use the **if not** statement, which will return a Boolean value. Empty strings in Python evaluate to **False** and strings that are not Empty evaluate to **True**.

```
str1 = ''

if not str1:
    print('This string is empty')
else:
    print('This string is NOT empty')
```

Output:

```
This string is empty
```

Example 2

```
str2 = 'string'

if not str1:
    print('This string is empty')
else:
    print('This string is NOT empty')
```

Output:

```
This string is NOT empty
```

32. Nested List and the Sum Function

You can flatten a nested list using the *sum()* function. Note that this will work on a two-dimensional nested list.

```
nested_list = [[2, 4, 6],[8, 10, 12]]

new_list = sum(nested_list,[])
print(new_list)
```

Output:

```
[2, 4, 6, 8, 10, 12]
```

Please note that this is not the mos efficient way to flatten a list. But its freaking cool to know it , right? 😊

Using reduce function

Here is another cool trick you can use to flatten a two-dimensional This time we use reduce from functools module.

```
from functools import reduce

nested_list = [[2, 4, 6],[8, 10, 12]]

new_list = reduce(lambda x, y: x+y, nested_list)
print(new_list)
```

Output:

```
[2, 4, 6, 8, 10, 12]
```

33. Checking if a File Exists

Using the os module, you can check if a file exists. The os module has an *exists()* function from the *path()* class that returns a Boolean value. If a file exists it will return True, and if not, it will return False. When working with files, it is important to check if a file exists before trying to run it, to avoid generating errors.

You can pass the file path or file name as an argument. If the file you are checking is in the same folder as your Python file, you can pass the file name as an argument.

In the example below, we use the file name as an argument because the file is in the same folder as the Python file.

```
import os.path

file = os.path.exists('file.txt')

if file:
    print('This file exist')
else:
    print('This file does Not exist')
```

Output:

```
This file does Not exist
```

34. Set Comprehension

Set comprehension is similar to list comprehension; the only difference is that it returns a set and not a list. You can use set comprehension on an iterable (list, tuples, set, etc).

Let's say you have a list of uppercase strings and you want to convert them into lowercase strings and remove duplicates; here is how you do it using set comprehension. Since sets are not ordered, the order of the items in the iterable will be changed. Sets do not allow duplicates, so only one 'PEACE' will be in the output set.

```
list1 = ['LOVE', 'HATE', 'WAR', 'PEACE', 'PEACE']
set1 = {word.lower()for word in list1}

print(set1)
Output:
{'love', 'peace', 'war', 'hate'}
```

Set properties

1. Sets are not ordered.
2. They are surrounded by curly brackets
3. Sets are mutable
4. Sets do not allow duplicates.
5. They can only hold immutable type objects. For example, you cannot put a list inside a set; it will generate an error. See below:

```
a = {[2,4]}
print(a)
Output:
TypeError: unhashable type: 'list'
```

35. Python *args and **Kwargs

When you are not sure how many arguments you will need for your function, you can pass ***args**(Non Keywords Arguments)* as a parameter. The * notation tells Python that you are not sure how many arguments you need and Python allows you to pass in as many arguments as you want. Below, we calculate average with different number of arguments

```
def avg(*args):
    avg1 = sum(args)/len(args)
    return f'The average is {avg1:.2f}'

print(avg(12, 34, 56))
print(avg(23,45,36,41,25,25))
```

Output:

```
The average is 34.00
The average is 32.50
```

When you see ****kwargs** (keywords arguments) as a parameter, it means the function can accept any number of arguments as a dictionary (arguments must be in key-value pairs). See the example below:

```
def myFunc(**kwargs):
    for key, value in kwargs.items():
        print(f'{key} = {value}')
    print('\n')

myFunc(Name = 'Ben',Age = 80, Occupation ='Engineer')
```

Output:

```
Name = Ben
Age = 80
Occupation = Engineer
```

36. Python Filter Function

You can use the filter function as an alternative to the *for loop*. If you want to return items from an iterable that match certain criteria, you can use the Python filter() function.

Let's say we have a list of names and we want to return a list of names that are lowercase; here is how you can do it using the filter() function.

The first example uses a for loop for comparison purposes.

```
for name in names:
    if name.islower():
        lower_case.append(name)
print(lower_case)
Output:
['moses', 'linda']
```

Example 2 (Using flter function)

```
names = ['Derick', 'John', 'moses', 'linda']

lower_case = list(filter(lambda x:x.islower(), names))
print(lower_case)
Output:
['moses', 'linda']
```

37. Dictionary Comprehension

Dictionary comprehension is a one-line code that transforms a dictionary into another dictionary with modified values. It makes your code intuitive and concise.

Let's say you want to update the values of the dictionary from ints to floats; you can use dictionary comprehension:

```
dict1 ={'Grade': 70, 'Weight': 45, 'Width': 89}

# Converting dict values into floats
dict2 = {k: float(v) for (k,v) in dict1.items()}
print(dict2)
Output:
{'Grade': 70.0, 'Weight': 45.0, 'Width': 89.0}
```

Dictionary properties

1. Dictionaries are surrounded by curly brackets
2. Dictionaries cannot have duplicate keys, but can have duplicate values
3. Dictionaries keys must be immutable
4. Dictionaries are now ordered(since Python 3.7). They remember the order of items inserted.

38. DataFrame from Two Lists

The easiest way to create a DataFrame from a two lists is to use the *pandas* module. Import pandas module and pass the lists to the DataFrame constructor. Since we have two lists, we have to use the *zip*() function to combine the lists.

Below, we have a list of car brands and a list of car models. We are going to create a DataFrame. The DataFrame will have one column called **Brands,** and another called **Models**, and the **index** will be the numbers in ascending order.

```
list1 = ['Tesla', 'Ford', 'Fiat']
models = ['X', 'Focus', 'Doblo']

df = pd.DataFrame(list(zip(list1,models)),
                  index =[1, 2, 3],
                  columns=['Brands','Models'])

print(df)
```

Output:

```
Brands  Models
1 Tesla     X
2  Ford   Focus
3  Fiat   Doblo
```

39. Adding a Column to a DataFrame

Let's continue with the tip from the previous question. Let's add a column called Age to the DataFrame. The column will have the ages of the cars.

```
import pandas as pd

list1 = ['Tesla', 'Ford', 'Fiat']
models = ['X', 'Focus', 'Doblo']

df = pd.DataFrame(list(zip(list1,models)),
                  index =[1, 2, 3],
                  columns=['Brands','Models'])

# Adding a column to dataFrame
df['Age'] = [2, 4, 3]

print(df)
```

Output:

```
Brands  Models  Age
1 Tesla    X      2
2  Ford   Focus   4
3  Fiat   Doblo   3
```

40. Code Timer as a Decorator

Below I created a timer function using the *perf_counter* class of the *time* module. Notice that the inner function is inside the timer function; this is because we are creating a decorator. This decorator is later used by the *range_tracker* function. The @timer right before the function means that the function is being decorated by another function. **To decorate a function is to improve or add extra functionality to that function without modifying it**. By using a decorator, we are able to add a timer to the *range_tracker* function. We are using this timer to check how long it takes to create a list from a range.

```
import time

def timer(func):
    def inner():
        start = time.perf_counter()
        func()
        end = time.perf_counter()
        print(f'Run time is {end-start:.2f} secs')
    return inner

@timer
def range_tracker():
    lst = []
    for i in range(10000000):
        lst.append(i**2)

range_tracker()
```

Output:

```
Run time is 10.25 secs
```

41. List Comprehension vs Generators

A generator is similar to a list comprehension, but instead of square brackets, you use parenthesis. Generators yield one item at a time, while list comprehension releases everything at a time. Below, I compare list comprehension to a generator in two categories:

1. Speed of execution.
2. Memory allocation.

Conclusion.

List comprehension executes much faster but takes up more memory. Generators execute a bit slower, but they take up less memory since they only yield one item at a time.

```
import timeit
import sys

def timer(_, code):
    exc_time = timeit.timeit(code, number=1000)
    return f'{_}: execution time is {exc_time:.2f}'

def memory_size(_, code):
    size = sys.getsizeof(code)
    return f'{_}: allocated memory is {size}'
one = 'Generator'
two = 'list comprehension'

print(timer(one, 'sum(num**2 for num in range(10000))'))
print(timer(two, 'sum([num**2 for num in range(10000)])'))
print(memory_size(one,(num**2 for num in range(10000))))
print(memory_size(two,[num**2 for num in range(10000)]))
```

Output:

```
Generator: execution time is 5.06
list comprehension: execution time is 4.60
Generator: allocated memory is 112
list comprehension: allocated memory is 85176
```

42. Writing to File

Let's say you have a list of names and you want to write them to a file and you want all the names to be written vertically. Here is a sample code that demonstrates how you can do it. The code below creates a CSV file. We use the escape character ('\n') to tell the code to write each name in a new line. Another way you can create a CSV file is by using the CSV module.

```
names = ['John Kelly', 'Moses Nkosi', 'Joseph Marley']

with open('names.csv', 'w') as file:
    for name in names:
        file.write(name)
        file.write('\n')

# reading CSV file
with open ('names.csv', 'r') as file:
    print(file.read())
```

Output:

```
John Kelly
Moses Nkosi
Joseph Marley
```

Using CSV Module

If you don't want to use this method, you can import the CSV module. CSV is a buit-in module that is come with Python, so no need to install it. Here is how you can use the CSV module to accomplish the same thing. Example code on the next page:

```
import csv

names = ['John Kelly', 'Moses Nkosi', 'Joseph Marley']

with open('names.csv', 'w') as file:
    for name in names:
        writer = csv.writer(file,lineterminator = '\n')
        writer.writerow([name])

# Reading the file
with open ('names.csv', 'r') as file:
    print(file.read())
```

Output:

```
John Kelly
Moses Nkosi
Joseph Marley
```

43. Merge PDF Files

If you want to merge PDF files you can do it in Python. Here is how you can do it using the PyPDF2 module. You can merge as many files as you want. You can install it using pip.

Pip install pyPDF2

```
import PyPDF2
from PyPDF2 import PdfFileMerger,
PdfFileReader,PdfFileWriter

list1 = ['file1.pdf', 'file2.pdf']


merge = PyPDF2.PdfFileMerger(strict=True)
for file in list1:
    merge.append(PdfFileReader(file,'rb+'))
merge.write('mergedfile.pdf')
merge.close()
created_file = PdfFileReader('mergedfile.pdf')
created_file
```

Output:

```
<PyPDF2.pdf.PdfFileReader at 0x257d1c8ba90>
```

44. Return vs Yield

The difference between the return statement and the yield statement.
The return statement returns one element and ends the function.
The yield statement returns a 'package' of elements called a generator. You have to 'unpack' the package to get the elements. You can use a for loop or the next() function to un-pack the generator.

Example 1 : Using return

```
def num (n: int) -> int:
    for i in range(n):
        return i


print(num(5))
```

Output:

```
0
```

Example 2 : Using yield

```
def num (n: int):
    for i in range(n):
        yield i

# creating a generator
gen = num(5)

#unpacking generator
for i in gen:
    print(i, end = ' ')
```

Output:

```
0 1 2 3 4
```

45. High Order Functions

A high order function is a function that takes another function as an argument, or returns another function. The code below demonstrates how we can create a function and use it inside a high order function. We create a function called *sort_names* and we use it as a key inside the *sorted()* function. By using index[0] we are basically telling the sorted function to sort names by their first name. If we use [1], then the names would be sorted by the last name.

```
def sort_names(x):
    return x[0]

names = [('John','Kelly'),('Chris','Rock'),
         ('Will','Smith')]

sorted_names = sorted(names, key= sort_names)
print(sorted_names)
```

Output:

```
[('Chris', 'Rock'), ('John', 'Kelly'), ('Will', 'Smith')]
```

46. Grammar Errors

Did you know that you can correct grammar errors in text using Python? You can use an open-source framework, Gramformer. Gramformer (created by Prithviraj Damodaran) is a framework for highlighting, and correcting grammar errors on natural language text. Here is a simple code that demonstrates how you can use gramformer. to correct errors in a text.

First, you need to install:

```
!pip3 install torch==1.8.2+cu111 torchvision==0
.9.2+cu111 torchaudio==0.8.2 -
f https://download.pytorch.org/whl/lts/1.8/torc
h_lts.html

!pip3 install -
U git+https://github.com/PrithivirajDamodaran/G
ramformer.git
```

```
from gramformer import Gramformer
# instantiate the model
gf = Gramformer(models=1, use_gpu=False)

sentences = [
             'I hates walking night',
             'The city is were I work',
             'I has two children'
              ]

for sentence in sentences:
    correct_sentences = gf.correct(sentence)
    print('[Original Sentence]', sentence)
    for correct_sentence in correct_sentences:
        print('[Corrected sentence]', correct_sentence)
```

Output:

```
[Original Sentence] I hates walking night
[Corrected sentence] I hate walking at night.
[Original Sentence] The city is were I work
[Corrected sentence] The city where I work.
[Original Sentence] I has two children
[Corrected sentence] I have two children.
```

47. Zen of Python

A Zen of Python is a list of 19 guiding principles for writing beautiful code. Zen of Python was written by Tim Peters and was later added to Python.

Here is how you can access the Zen of Python.

```
import this

print(this)
```

Output:

```
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

48. Sorted by Pprint

Do you know that you can print out a sorted dictionary using pprint module? Below we use the module to print out a dictionary in ascending order.

```
import pprint

a = {'c': 2, 'b': 3, 'y': 5,'x': 10}
pp = pprint.PrettyPrinter(sort_dicts=True)
pp.pprint(a)
Output:
{'b': 3, 'c': 2, 'x': 10, 'y': 5}
```

Please note that Pprint does not change the order of the actual dictionary; it just changes the printout order.

49. Convert Picture to gray Scale

Do you want to convert a color image into grayscale? Use Python's cv2 module.

First install cv2 using > **pip install opencv-python**<

Below we are converting a color book image to grayscale. You can replace that with your own image. You must know where the image is stored.

When you want to view an image using CV2 a window will open. The wait key() in milliseconds, is the time you want the window to stay open before it closes.

```
import cv2 as cv


img = cv.imread('book.jpg')

# show the original image
img1 = cv.imshow('Original', img)
cv.waitKey(5)

# Converting image to Gray
grayed_img = cv.cvtColor(img, cv.COLOR_BGR2GRAY)

# Show grayed-out image
img2 = cv.imshow('grayed_image', grayed_img)
cv.waitKey(5000)

#Save image
cv.imwrite('grayed.jpg', grayed_img)
```

50. Time it with timeit

If you want to know the execution time of a given code, you can use the *timeit()* method. Below, we create a timer function, with the timeit function. The timeit function takes the code we want to check and the number of executions we want to run(number=1000). When we call the function, We then pass a code to the function to determine execution time. Basically, below we are using timeit to check how long it takes to execute **sum(num**2 for num in range(10000)).**

```
import timeit

def timer(code):
    tm = timeit.timeit(code,number=1000)
    return f'Execution time is {tm:.2f} secs.'

if __name__ == "__main__":
    print(timer('sum(num**2 for num in range(10000))'))
```

Output:

```
Execution time is 5.05 secs.
```

You can also use *timeit* without creating a function. Remember that the *stmt* parameter only takes a string as an argument; that is why the *test* variable below, which is the code that we pass to the timeit method, is a string.

```
import timeit

test = "sum(num ** 2 for num in range(10000))"
tm = timeit.timeit(stmt=test,number=1000)
print(f'{tm:.2f} secs')
```

Output

```
2.20 secs
```

Other Books by author

Practice Python : Questions and Challenges for Beginners

Ohttps://benjaminb.gumroad.com/l/ggTvR

50 Days of Python: A Challenge a Day

https://benjaminb.gumroad.com/l/zybjn

Master Python Fundementals: The Ultimate Guide for Beginners

Coming soon

The End