100 + Python Tips

(with Code)

by Chetan Kandpal

SkyRocket your Machine Learning skills by applying these basic Python tips.

swipe >

Inspired by the article by **Fatos Morina**.
Link: *https://towardsdatascience.com/100-helpful-python-tips-you-can-learn-before-finishing-your-morning-coffee-eb9c39e68958*

Do you want to learn Python, SQL, Django, Machine Learning, Deep Learning, and Statistics in one-on-one classes
Drop me a message on LinkedIn to discuss your requirements

0. Get Python Version in Jupyter Notebook

```
In [32]:from platform import python_version

print(python_version())
```

3.8.13

1. “Else” condition inside a “for” loop

Despite all the Python code that you have seen so far, chances are that you may have missed the following “for-else” which I also got to see for the first time a couple of weeks ago.

This is a “for-else” method of looping through a list, where despite having an iteration through a list, you also have an “else” condition, which is quite unusual.

This is not something that I have seen in other programming languages like Java, Ruby, or JavaScript.

Let’s see an example of how it looks in practice.

Let’s say that we are trying to check whether there are no odd numbers in a list.

Let’s iterate through it:

```
In [1]:numbers = [2, 4, 6, 8, 1]

for number in numbers:
if number % 2 == 1:
print(number)
break
else:
                print("No odd numbers")
```

1

In case we find an odd number, then that number will be printed since break will be

executed and the else branch will be skipped.

Otherwise, if break is never executed, the execution flow will continue with the else branch.

In this example, we are going to print 1.

If we remove all odd numbers

```
In [9]:numbers = [2, 4, 6, 8]

for number in numbers:
if number % 2 == 1:
print(number)
break
else:
                print("No odd numbers")
```

No odd numbers

2. Get elements from a list using named variables

```
In [2]:my_list = [1, 2, 3, 4, 5]
       one, two, three, four, five = my_list
```

```
In [3]:one
```

Out[3]: 1

```
In [4]:two
```

Out[4]: 2

```
In [5]:three
```

Out[5]: 3

```
In [6]:four
```

Out[6]: 4

```
In [7]:five
```

Out[7]: 5

3. Get n largest or n smallest elements in a list using the module heapq

```
In [10]:import heapq

         scores = [51, 33, 64, 87, 91, 75, 15, 49, 33, 82]
```

```
print(heapq.nlargest(3, scores)) # [91, 87, 82]
print(heapq.nsmallest(5, scores)) # [15, 33, 33, 49, 51]
```

```
[91, 87, 82]
[15, 33, 33, 49, 51]
```

4. Pass values from a list as method arguments

We can extract all elements of a list using “*”:

```
In [12]:my_list = [1, 2, 3, 4]
                print(my_list) # [1, 2, 3, 4]
print(*my_list) # 1 2 3 4
```

```
[1, 2, 3, 4] 1 2
3 4
```

This can be helpful in situations where we want to pass all elements of a list as method arguments:

```
In [13]:def sum_of_elements(*arg):
total = 0
for i in arg:
total += i

return total


      result = sum_of_elements(*[1, 2, 3, 4])
print(result) # 10
```

```
10
```

5. Get all the elements in the middle of the list

```
In [14]:_, *elements_in_the_middle, _ = [1, 2, 3, 4, 5, 6, 7, 8]
print(elements_in_the_middle) # [2, 3, 4, 5, 6, 7]
```

```
[2, 3, 4, 5, 6, 7]
```

6. Assign multiple variables in just one line

```
In [15]:one, two, three, four = 1, 2, 3, 4
```

```
In [16]:one
```

```
Out[16]: 1
```

```
In [17]:two
```

```
Out[17]: 2
```

```
In [18]:three
```

```
Out[18]: 3
```

```
In [19]:four
```

```
Out[19]: 4
```

7. List comprehensions

You can loop through the elements in a list in a single line in a very comprehensive way.

Let's see that in action in the following example:

```
In [20]:numbers = [1, 2, 3, 4, 5, 6, 7, 8]

        even_numbers = [number for number in numbers if number % 2 == 0]

print(even_numbers) # [2, 4, 6, 8]
```

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

We can do the same with dictionaries, sets, and generators.

Let's see another example, but now for dictionaries.

```
In [21]:dictionary = {'first_element': 1, 'second_element': 2,
'third_element': 3, 'fourth_element': 4}
odd_value_elements = {key: num for (key, num) in
dictionary.items() if num % 2 == 1}
                print(odd_value_elements) # {'first_element': 1, 'third_element': 3}
```

```
{'first_element': 1, 'third_element': 3}
```

8. Enumerate related items of the same concept via Enum

An Enum is a set of symbolic names bound to unique values. They are similar to global variables, but they offer a more useful repr(), grouping, type-safety, and a few other features.

```
In [22]:from enum import Enum

class Status(Enum):
NO_STATUS = -1
NOT_STARTED = 0
IN_PROGRESS = 1
COMPLETED = 2


        print(Status.IN_PROGRESS.name) # IN_PROGRESS
print(Status.COMPLETED.value) # 2
```

IN_PROGRE
SS 2

9. Repeat strings without looping

In [23]:
```
string = "Abc"

print(string * 5) # AbcAbcAbcAbcAbc
```

AbcAbcAbcAbcAbc

10. Compare 3 numbers just like in Math

If you have a value and you want to compare it whether it is between two other values, there is a simple expression that you use in Math:

1 < x < 10

That is the algebraic expression that we learn in elementary school. However, you can also use that same expression in Python as well.

Yes, you heard that right. You have probably done comparisons of such form up until now:

1 < x and x < 10

For that, you simply need to use the following in Python:

1 < x < 10

In [25]:
```
x=3

print(1<x<10)
```

True

This doesn't work in Ruby, the programming language that was developed with the intention of making programmers happy. This turns out to be working in JavaScript as well.

I was really impressed seeing such a simple expression not being talked about more widely. At least, I haven't seen it being mentioned that much.

11. Merge dictionaries in a single readable line

This is available as of Python 3.9:

first_dictionary = {'name': 'Fatos', 'location': 'Munich'} second_dictionary = {'name': 'Fatos', 'surname': 'Morina','location': 'Bavaria, Munich'} result = first_dictionary | second_dictionary print(result) {'name': 'Fatos', 'location': 'Bavaria, Munich', 'surname': 'Morina'}

12. Find the index of an element in a tuple

```
In [1]:books = ('Atomic habits', 'Ego is the enemy', 'Outliers', 'Mastery')

print(books.index('Mastery'))
```

3

13. Convert a string into a list of strings

Let's say that you get the input in a function that is a string, but it is supposed to be a list:

input = "[1,2,3]"

You don't need it in that format. You need it to be a list:

input = [1,2,3]

Or maybe you the following response from an API call:

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

Rather than bothering with complicated regular expressions, all you have to do is import the module ast and then call its method literal_eval:

```
In [2]:import ast


    def string_to_list(string):
            return ast.literal_eval(string)
```

That's all you need to do.

Now you will get the result as a list, or list of lists, namely like the following:

```
    In [3]:string = "[[1, 2, 3],[4, 5, 6]]"
     my_list = string_to_list(string)
print(my_list)
```

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

we can also use lambda function

```
In [4]:string_to_list = lambda string: ast.literal_eval(string)
```

14. Avoid "trivial" mistakes by using named parameters

Let's assume that you want to find the difference between 2 numbers. The difference is not commutative:

a - b != b -a

However, we may forget the ordering of the parameters which can cause "trivial"

mistakes:

```
In [5]:def subtract(a, b):
return a - b


      print((subtract(1, 3)))
     print((subtract(3, 1)))
```

```
-2
2
```

To avoid such potential mistakes, we can simply use named parameters and the ordering of parameters doesn't matter anymore:

```
In [6]:def subtract(a, b):
        return a - b


print((subtract(a=1, b=3)))
print((subtract(b=3, a=1)))
```

```
-2
-2
```

15. Print multiple elements with a single print() statement

```
In [7]:print(1, 2, 3, "a", "z", "this is here", "here is something else")
```

```
1 2 3 a z this is here here is something else
```

16. Print multiple elements in the same line

```
In [8]:print("Hello", end="")
print("World") # HelloWorld
print("Hello", end=" ")
print("World") # Hello World
                print('words', 'with', 'commas', 'in', 'between', sep=', ')
```

```
HelloWorld
Hello World
words, with, commas, in, between
```

17. Print multiple values with a custom separator in between each value

You can do advanced printing quite easily:

```
In [9]:print("name", "domain.com", sep="@")

day=29
month=1
```

```
year=2022
print(day,month, year,sep="/")
```

```
name@domain.c
om 29/1/2022
```

18. You cannot use a number at the beginning of the name of a variable

```
In [10]:four_letters = "abcd" # this works
```

```
In [11]:4_letters = "abcd"# this doesn't work
```

```
 Input In [11]
        4_letters = "abcd"# this doesn't work
^
SyntaxError: invalid decimal literal
```

19. You cannot use an operator at the beginning of the name of a variable

```
In [12]:+variable = "abcd" # this doesn't work
```

```
 Input In [12]
+variable = "abcd" # this doesn't work
^
SyntaxError: invalid character in identifier
```

20. You cannot have 0 as the first digit in a number

```
In [13]:number = 0110 # this doesn't work
```

```
 Input In [13]
number = 0110 # this doesn't work
^
SyntaxError: leading zeros in decimal integer literals are not permitted; u se an 0o prefix for octal integers
```

21. You can use the underscore character anywhere in the name of a variable

This means, anywhere you want, as many times as you want in the name of a variable:

```
In [14]:a_____b = "abcd" # this works
```

```
In [15]:_a_b_c_d = "abcd" # this also works
```

I am not encouraging you to use it, but in case you see a weird naming of a variable

like that, know that it is actually a valid name of a variable

22. You can separate big numbers with the underscore

This way it can be easier to read them.

```
In [16]:print(1_000_000_000)
print(1_234_567)
```

```
100000000
0 1234567
```

23. Reverse the ordering of a list

```
In [17]:my_list = ['a', 'b', 'c', 'd']

my_list.reverse()

                print(my_list) # ['d', 'c', 'b', 'a']
```

```
['d', 'c', 'b', 'a']
```

24. Slice a string using a step function

```
In [18]:my_string = "This is just a sentence"
print(my_string[0:5]) # This

# Take three steps forward
print(my_string[0:10:3]) # Tsse
```

```
This
Tssu
```

25. Reverse slicing

```
In [19]:my_string = "This is just a sentence"
          print(my_string[10:0:-1]) # suj si sih

# Take two steps forward
print(my_string[10:0:-2]) # sjs i
```

```
suj  si  sih
sjs i
```

26. Partial Slicing with just the beginning or ending index

Indices indicating the start and end of slicing can be optional.

```
In [20]:my_string = "This is just a sentence"
```

```
print(my_string[4:])  # is just a sentence

print(my_string[:3]) # Thi
```

```
 is just a sentence
Thi
```

27. Floor division

```
In [21]:print(3/2) # 1.5
print(3//2) # 1
```

```
1.5 1
```

28. Difference between == and “is”

“is” checks whether 2 variables are pointing to the same object in memory.

“==” compares the equality of values that these 2 objects hold.

```
In [22]:first_list = [1, 2, 3]
second_list = [1, 2, 3]

# Is their actual value the same?
print(first_list == second_list) # True

# Are they pointing to the same object in memory
print(first_list is second_list)
      # False, since they have same values, but in different objects in memory


third_list = first_list

print(third_list is first_list)
# True, since both point to the same object in memory
```

```
True
False
True
```

29. Merge 2 dictionaries quickly

```
In [23]:dictionary_one = {"a": 1, "b": 2}
dictionary_two = {"c": 3, "d": 4}

merged = {**dictionary_one, **dictionary_two}

                print(merged) # {'a': 1, 'b': 2, 'c': 3, 'd': 4}
```

```
{'a': 1, 'b': 2, 'c': 3, 'd': 4}
```

30. Check whether a string is larger than another string

```
In [24]:first = "abc"
second = "def"
print(first < second) # True
second = "ab"
      print(first < second) # False
```

```
True
False
```

31. Check whether a string starts with a particular character without using the index 0

```
In [25]:my_string = "abcdef"
       print(my_string.startswith("b")) # False
```

```
False
```

32. Find the unique id of a variable using id()

```
In [26]:print(id(1))
print(id(2))
     print(id("string"))
```

```
94607791860992
94607791861024
140279240972016
```

33. Integers, floats, strings, booleans, and tuples are immutable

When we assign a variable to an immutable type such as integers, floats, strings, booleans, and tuples, then this variable points to an object in memory.

In case we assign to that variable another value, the original object is still in memory, but the variable pointing to it is lost:

```
In [27]:number = 1
print(id(number))
print(id(1))

number = 3
print(id(number))
print(id(1))
```

```
9460779186099
2
9460779186099
2
9460779186105
6
9460779186099
2
```

34. Strings and tuples are immutable

This was already mentioned in the previous point but wanted to emphasize it since this is quite important.

```
In [28]:name = "Fatos"
print(id(name))
name = "fatos"
print(id(name))
```

```
140279140850416
140279140584688
```

```
In [29]:my_tuple = (1, 2, 3, 4)
print(id(my_tuple))

my_tuple = ('a', 'b')
print(id(my_tuple))
```

```
140279140199136
140279140647104
```

35. Lists, sets, and dictionaries are mutable

This means that we can change the object without losing binding to it:

```
In [30]:cities = ["Munich", "Zurich", "London"]
print(id(cities))

cities.append("Berlin")
print(id(cities))
```

```
140279140132864
140279140132864
```

Here is another example with sets:

```
In [31]:my_set = {1, 2, 3, 4}
print(id(my_set))
my_set.add(5)
print(id(my_set))
```

```
140279166983520
140279166983520
```

36. You can turn a set into an immutable set

This way, you can no longer modify it:

If you do that, an error will be thrown:

```
In [32]:my_set = frozenset(['a', 'b', 'c', 'd'])

my_set.add("a")
```

```
---------------------------------------------------------------------------
Traceback (most recent call last) Input In [32], in <cell line: 3>()
1 my_set = frozenset(['a', 'b', 'c', 'd'])
----> 3 my_set.add("a")

AttributeError: 'frozenset' object has no attribute 'add'
```

AttributeError

37. An “if-elif” block can exist without the else block at the end

However, “elif” cannot stand on its own without an “if” step before it:

```
In [33]:def check_number(number):
if number > 0:
                    return "Positive"
elif number == 0:
return "Zero"

return "Negative"


print(check_number(1))
```

```
Positive
```

38. Check whether 2 strings are anagrams using sorted()

```
In [34]:def check_if_anagram(first_word, second_word):
first_word = first_word.lower()
second_word = second_word.lower()
return sorted(first_word) == sorted(second_word)
      print(check_if_anagram("testinG", "Testing")) # True
print(check_if_anagram("Here", "Rehe")) # True
print(check_if_anagram("Know", "Now")) # False
```

```
True
True
False
```

39. Get the value of a character in Unicode

```
In [35]:print(ord("A")) # 65
     print(ord("B")) # 66
     print(ord("C")) # 66
    print(ord("a")) # 97
```

```
65
66
67
97
```

40. Get keys of a dictionary in a single line

```
In [36]:dictionary = {"a": 1, "b": 2, "c": 3}

keys = dictionary.keys()

print(list(keys)) # ['a', 'b', 'c']
```

```
['a', 'b', 'c']
```

41. Get values of a dictionary in a single line

```
In [37]:dictionary = {"a": 1, "b": 2, "c": 3}

values = dictionary.values()

print(list(values)) # [1, 2, 3]
```

```
[1, 2, 3]
```

42. Swap keys and values of a dictionary

```
In [38]:dictionary = {"a": 1, "b": 2, "c": 3}

reversed_dictionary = {j: i for i, j in dictionary.items()}

print(reversed) # {1: 'a', 2: 'b', 3: 'c'}
```

```
<class 'reversed'>
```

```
In [39]:reversed_dictionary
```

```
Out[39]: {1: 'a', 2: 'b', 3: 'c'}
```

43. You can convert a boolean value into a number

```
In [40]:print(int(False)) # 0
print(float(True)) # 1.0
```

```
0
1.0
```

44. You can use boolean values in arithmetic operations

“False” is 0, whereas “True” is 1.

```
In [41]:x = 10
y = 12

result = (x - False)/(y * True)

print(result)
```

```
0.8333333333333334
```

45. You can convert any data type into a boolean value

```
In [42]:print(bool(.0)) # False
print(bool(3)) # True
print(bool("-")) # True
        print(bool("string")) # True
print(bool(" ")) # True
```

```
False
True
True
True
True
```

46. Convert a value into a complex number

```
In [43]:print(complex(10, 2))
```

```
(10+2j)
```

You can also convert a number into a hexadecimal number:

```
In [44]:print(hex(11))
```

```
0xb
```

47. Add a value in the first position in a list

If you use append(), you are going to insert new values from the right.

We can also use insert() to specify the index and the element where we want to insert this new element. In our case, we want to insert it in the first position, so we use 0 as the index:

```
In [45]:my_list = [3, 4, 5]

my_list.append(6)
        my_list.insert(0, 2)
print(my_list)
```

```
[2, 3, 4, 5, 6]
```

48. Lambda functions can only be in one line

You cannot have lambdas in more than one line.

Let's try the following:

```
In [47]:comparison = lambda x: if x > 3:
                                   print("x > 3")
```

```
else:
                                    print("x is not greater than 3")
```

```
  File <tokenize>:3
else:
^
IndentationError: unindent does not match any outer indentation level
```

Correct way

```
In [48]:comparison = lambda x: print("x > 3") if x > 3 else print("x is not greater
```

```
In [49]:result = lambda x: if x > 3:
```

```
  Input In [49]
          result = lambda x: if x > 3:
^
SyntaxError: invalid syntax
```

Correct way

```
In [50]:result = lambda x: x > 3
```

49. Conditionals statements in lambda should always include the “else” part

Let’s try the following:

```
In [51]:comparison = lambda x: "x > 3" if x > 3
```

```
  Input In [51]
comparison = lambda x: "x > 3" if x > 3
                                        ^
SyntaxError: invalid syntax
```

that this is a feature of the conditional expression and not of the lambda itself.

50. filter() returns a new object

```
In [52]:my_list = [1, 2, 3, 4]

        odd = filter(lambda x: x % 2 == 1, my_list)

print(list(odd))
print(my_list)
```

```
[1, 3]
[1, 2, 3, 4]
```

51. map() returns a new object

```
In [53]:my_list = [1, 2, 3, 4]
```

```
squared = map(lambda x: x ** 2, my_list)

print(list(squared)) # [1, 4, 9, 16]
print(my_list) # [1, 2, 3, 4]
```

[1, 4, 9, 16] [1,
2, 3, 4]

52. range() includes a step parameter that may not be known that much

```
In [54]:for number in range(1, 10, 3):
print(number, end=" ")
# 1 4 7
```

1 4 7

53. range() starts by default at 0

So you don't need to include it at all.

```
In [55]:def range_with_zero(number):
for i in range(0, number):
print(i, end=' ')


          def range_with_no_zero(number):
for i in range(number):
print(i, end=' ')


range_with_zero(3) # 0 1 2
range_with_no_zero(3) # 0 1 2
```

0 1 2 0 1 2

54. You don't need to compare the length with 0

If the length is greater than 0, then it is by default True, so you don't really need to compare it with 0:

```
In [56]:def get_element_with_comparison(my_list):
if len(my_list) > 0:
return my_list[0]


def get_first_element(my_list):
if len(my_list):
return my_list[0]


elements = [1, 2, 3, 4]
first_result = get_element_with_comparison(elements)
          second_result = get_element_with_comparison(elements)

             print(first_result == second_result) # True
```

True

55. You can define the same method multiple times inside the same scope

However, only the last one is called, since it overrides previous ones.##

```
In [57]:def get_address():
return "First address"


def get_address():
return "Second address"


def get_address():
return "Third address"


    print(get_address()) # Third address
```

```
Third address
```

56. You can access private properties even outside their intended scope

```
In [58]:class Engineer:
def __init__(self, name):
self.name = name
self.__starting_salary = 62000


dain = Engineer('Dain')
    print(dain._Engineer__starting_salary) # 62000
```

```
62000
```

57. Check the memory usage of an object

```
In [59]:import sys


      print(sys.getsizeof("bitcoin")) # 56
```

```
56
```

58. You can define a method that can be called with as many parameters as you want

```
In [60]:def get_sum(*arguments):
result = 0
for i in arguments:
result += i
return result


print(get_sum(1, 2, 3)) # 6
print(get_sum(1, 2, 3, 4, 5)) # 15
          print(get_sum(1, 2, 3, 4, 5, 6, 7)) # 28
```

```
6
15
28
```

59. You can call the parent class's initializer using super() or parent class's name

Calling the parent's class initializer using super():

```
In [61]:class Parent:
def __init__(self, city, address):
self.city = city
self.address = address


class Child(Parent):
def __init__(self, city, address, university):
                    super().__init__(city, address)
self.university = university


                child = Child('Zürich', 'Rämistrasse 101', 'ETH Zürich')
print(child.university) # ETH Zürich
```

```
ETH Zürich
```

Calling the parent's class using the parent class's name:

```
In [62]:class Parent:
def __init__(self, city, address):
self.city = city
self.address = address


class Child(Parent):
def __init__(self, city, address, university):
Parent.__init__(self, city, address)
self.university = university


                child = Child('Zürich', 'Rämistrasse 101', 'ETH Zürich')
print(child.university) # ETH Zürich
```

```
ETH Zürich
```

Note that calls to parent initializers using **init**() and super() can only be used inside the child class's initializer.

60. You can redefine the "+" operator inside your own classes

Whenever you use the + operator between two int data types, then you are going to find their sum.

However, when you use it between two string data types, you are going to merge them:

```
In [63]:print(10 + 1) # Adding two integers using '+'
        print('first' + 'second') # Merging two strings '+'
```

```
11
firstsecond
```

This represents the operator overloading.

You can also use it with your own classes as well:

```
In [64]:class Expenses:
def __init__(self, rent, groceries):
self.rent = rent
self.groceries = groceries

def __add__(self, other):
return Expenses(self.rent + other.rent,
                                    self.groceries + other.groceries)


april_expenses = Expenses(1000, 200)
may_expenses = Expenses(1000, 300)

total_expenses = april_expenses + may_expenses
print(total_expenses.rent) # 2000
print(total_expenses.groceries) # 500
```

```
2000
500
```

61. You can also redefine the “<” and “==” operators inside your own classes

Here is another example of an operation overloadding that you can define yourself:

```
In [65]:class Game:
def __init__(self, score):
self.score = score

def __lt__(self, other):
                    return self.score < other.score


first = Game(1)
second = Game(2)

    print(first < second) # True
```

```
True
```

we can override the **eq**() function based on our own needs:

```
In [66]:class Journey:
def __init__(self, location, destination, duration):
self.location = location
self.destination = destination
self.duration = duration

def __eq__(self, other):
return ((self.location == other.location) and
                            (self.destination == other.destination) and
```

```
 (self.duration == other.duration))


first = Journey('Location A', 'Destination A', '30min')
second = Journey('Location B', 'Destination B', '30min')

print(first == second)
```

```
False
```

You can also analogously define:

sub() for -

mul() for *

truediv() for /

ne() for !=

ge() for >=

gt() for >

62. You can define a custom printable version for an object of a class

```
In [67]:class Rectangle:
def __init__(self, a, b):
 self.a = a
self.b = b

def __repr__(self):
                                return repr('Rectangle with area=' + str(self.a * self.b))


print(Rectangle(3, 4))
```

```
'Rectangle with area=12'
```

63. Swap cases of characters in a string

```
    In [68]:string = "This is just a sentence."
      result = string.swapcase()
print(result)
```

```
tHIS IS JUST A SENTENCE.
```

64. Check if all characters are white spaces in a string

```
In [69]:string = " "
     result = string.isspace()
print(result)
```

```
True
```

65. Check if all characters in a string are either alphabets or numbers

```
In [70]:name = "Password"
print(name.isalnum()) # True, because all characters are alphabets
name = "Secure Password "
print(name.isalnum()) # False, because it contains whitespaces
name = "S3cur3P4ssw0rd"
print(name.isalnum()) # True
name = "133"
print(name.isalnum()) # True, because all characters are numbers
```

```
True
False
True
True
```

66. Check if all characters in a string are alphabets

```
In [71]:string = "Name"
print(string.isalpha()) # True
string = "Firstname Lastname"
print(string.isalpha()) # False, because it contains whitespace
string = "P4ssw0rd"
print(string.isalpha()) # False, because it contains numbers
```

```
True
False
False
```

67. Remove characters from the right based on the argument

```
In [72]:string = "This is a sentence with "
# Remove trailing spaces from the right
print(string.rstrip()) # "This is a sentence with"
string = "this here is a sentence.....,,,,aaaaasd"
print(string.rstrip(".,dsa")) # "this here is a sentence"
```

```
This is a sentence with
this here is a sentence...
```

You can similarly remove characters from the left based on the argument:

```
In [73]:string = "ffffffffFirst"
print(string.lstrip("f")) # First
```

```
First
```

68. Check if a string represents a number

```
In [74]:string = "seven"
print(string.isdigit()) # False
string = "1337"
print(string.isdigit()) # True
string = "5a"
```

```
print(string.isdigit()) # False, because it contains the character 'a' string =
"2**5"
print(string.isdigit()) # False
```

```
False
True
False
False
```

69. Check if a string represents a Chinese number

```
In [75]:# 42673 in Arabic numerals
string = "四二六七三"  四二六七三

print(string.isdigit()) # False
      print(string.isnumeric()) # True
```

```
False
True
```

70. Check if a string has all its words starting with an uppercase character

```
In [76]:string = "This is a sentence"
print(string.istitle()) # False

string = "10 Python Tips"
print(string.istitle()) # True

string = "How to Print A String in Python"
          # False, because of the first characters being lowercase in "to" and "in"
print(string.istitle())

string = "PYTHON"
print(string.istitle()) # False. It's titlelized version is "Python"
```

```
False
True
False
False
```

71. Condition inside the print function

```
In [77]:def is_positive(number):
             print("Positive" if number > 0 else "Negative") # Positive


is_positive(-3)
```

```
Negative
```

72. Multiple conditions at a single if-statement

```
In [78]:math_points = 51
  biology_points = 78
  physics_points = 56
  history_points = 72
```

```
my_conditions = [math_points > 50, biology_points > 50,
                 physics_points > 50, history_points > 50]

if all(my_conditions):
print("Congratulations! You have passed all of the exams.")
else:
        print("I am sorry, but it seems that you have to repeat at least one exa
```

```
Congratulations! You have passed all of the exams.
```

73. At least one condition is met out of many in a single if-statement

```
In [79]:math_points = 51
biology_points = 78
physics_points = 56
history_points = 72

my_conditions = [math_points > 50, biology_points > 50,
physics_points > 50, history_points > 50]

if any(my_conditions):
print("Congratulations! You have passed all of the exams.")
else:
        print("I am sorry, but it seems that you have to repeat at least one exa
# Congratulations! You have passed all of the exams.
```

```
Congratulations! You have passed all of the exams.
```

74. Any non-empty string is evaluated to True

```
In [80]:print(bool("Non empty")) # True
print(bool("")) # False
```

```
True
False
```

75. Any non-empty list, tuple, or dictionary is evaluated to True

```
In [81]:print(bool([])) # False
        print(bool(set([]))) # False

print(bool({})) # False
        print(bool({"a": 1})) # True
```

```
False
False
False
True
```

76. Other values that evaluate to False are None, “False” and the number 0

```
In [82]:print(bool(False)) # False
print(bool(None)) # False
print(bool(0)) # False
```

```
False
False
False
```

77. You cannot change the value of a global variable just by mentioning it inside a function

```
In [83]:string = "string"

def do_nothing():
    string = "inside a method"


do_nothing()

print(string) # string
```

```
string
```

You need to use the access modifier global as well:

```
In [84]:string = "string"

def do_nothing():
global string
string = "inside a method"


do_nothing()

    print(string) # inside a method
```

```
inside a method
```

78. Count the number of elements in a string or list using Counter from “collections”

```
In [85]:from collections import Counter

result = Counter("Banana")
print(result) # Counter({'a': 3, 'n': 2, 'B': 1})


result = Counter([1, 2, 1, 3, 1, 4, 1, 5, 1, 6])
                    print(result) # Counter({1: 5, 2: 1, 3: 1, 4: 1, 5: 1, 6: 1})
```

```
Counter({'a': 3, 'n': 2, 'B': 1})
Counter({1: 5, 2: 1, 3: 1, 4: 1, 5: 1, 6: 1})
```

79. Check if 2 strings are anagrams using Counter

```
In [86]:from collections import Counter


    def check_if_anagram(first_string, second_string):
first_string = first_string.lower()
```

```
    second_string = second_string.lower()
    return Counter(first_string) == Counter(second_string)


print(check_if_anagram('testinG', 'Testing')) # True
print(check_if_anagram('Here', 'Rehe')) # True
print(check_if_anagram('Know', 'Now')) # False
```

```
True
True
False
```

You can also check whether 2 strings are anagrams using sorted():

```
In [87]:def check_if_anagram(first_word, second_word):
first_word = first_word.lower()
second_word = second_word.lower()
return sorted(first_word) == sorted(second_word)
print(check_if_anagram("testinG", "Testing")) # True
print(check_if_anagram("Here", "Rehe")) # True
print(check_if_anagram("Know", "Now")) # False
```

```
True
True
False
```

80. Count the number of elements using “count” from “itertools”

```
In [88]:from itertools import count

my_vowels = ['a', 'e', 'i', 'o', 'u', 'A', 'E', 'I', 'O', 'U']

current_counter = count()

string = "This is just a sentence."

for i in string:
if i in my_vowels:
print(f"Current vowel: {i}")
print(f"Number of vowels found so far: {next(current_counter)}")
```

```
Current vowel: i
Number of vowels found so far: 0
Current vowel: i
Number of vowels found so far: 1
Current vowel: u
Number of vowels found so far: 2
Current vowel: a
Number of vowels found so far: 3
Current vowel: e
Number of vowels found so far: 4
Current vowel: e
Number of vowels found so far: 5
Current vowel: e
Number of vowels found so far: 6
```

81. We can use negative indexes in tuples too

```
In [89]:numbers = (1, 2, 3, 4)

print(numbers[-1])
print(numbers[-4])
```

```
4
1
```

82. Nest a list and a tuple inside a tuple

```
In [90]:mixed_tuple = (("a"*10, 3, 4), ['first', 'second', 'third'])

print(mixed_tuple[1])
print(mixed_tuple[0])
```

```
['first',       'second',       'third']
('aaaaaaaaaa', 3, 4)
```

83. Quickly count the number of times an element appears in a list that satisfies a condition

```
In [91]:names = ["Besim", "Albert", "Besim", "Fisnik", "Meriton"]

print(names.count("Besim"))
```

```
2
```

84. You can easily get the last n elements using slice()

```
In [92]:my_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
slicing = slice(-4, None)
# Getting the last 3 elements from the list
print(my_list[slicing]) # [4, 5, 6]
      # Getting only the third element starting from the right
print(my_list[-3])
```

```
[7, 8, 9, 10] 8
```

You can also use slice() for other usual slicing tasks, like:

```
In [93]:string = "Data Science"
        # start = 1, stop = None (don't stop anywhere), step = 1
# contains 1, 3 and 5 indices
slice_object = slice(5, None)
print(string[slice_object])
```

```
Science
```

85. Count the number of times an element appears in a tuple

```
In [94]:my_tuple = ('a', 1, 'f', 'a', 5, 'a')

print(my_tuple.count('a'))
```

```
3
```

86. Get the index of an element in a tuple

```
In [95]:my_tuple = ('a', 1, 'f', 'a', 5, 'a')
print(my_tuple.index('f'))
```

```
2
```

87. Get sub-tuples by making jumps

```
In [96]:my_tuple = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
print(my_tuple[::3])
```

```
(1, 4, 7, 10)
```

88. Get sub-tuples starting from an index

```
In [97]:my_tuple = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
print(my_tuple[3:])
```

```
(4, 5, 6, 7, 8, 9, 10)
```

89. Remove all elements from a list, set, or dictionary

```
In [98]:my_list = [1, 2, 3, 4]
my_list.clear()
print(my_list) # []

my_set = {1, 2, 3}
my_set.clear()
print(my_set) # set()

        my_dict = {"a": 1, "b": 2}
my_dict.clear()
print(my_dict)
```

```
[]
set()
{}
```

90. Join 2 sets

One way is to use the method union() which returns a new set as a result of the joining:

```
In [99]:first_set = {4, 5, 6}
second_set = {1, 2, 3}

    print(first_set.union(second_set))
```

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

Another one is method update, which inserts the element of the second set into the first one:

```
In [100...first_set = {4, 5, 6}
second_set = {1, 2, 3}

    first_set.update(second_set)

print(first_set)
```

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

91. Sort elements of a string or list based on their frequency

Counter from the collections module by default doesn't order elements based on their frequencies.

```
In [102...from collections import Counter

        result = Counter([1, 2, 3, 2, 2, 2, 2])
print(result)
print(result.most_common())
```

```
Counter({2: 5, 1: 1, 3: 1}) [(2, 5),
(1, 1), (3, 1)]
```

92. Find the most frequent element in a list in just one line

```
In [103...my_list = ['1', 1, 0, 'a', 'b', 2, 'a', 'c', 'a']

print(max(set(my_list), key=my_list.count))
```

```
a
```

93. Difference between copy() and deepcopy()

Here is the explanation from the docs:
https://docs.python.org/3/library/copy.html#module-copy

A shallow copy constructs a new compound object and then (to the extent possible) inserts references into it to the objects found in the original.

A deep copy constructs a new compound object and then, recursively, inserts copies into it of the objects found in the original.

Maybe, an even more comprehensive description can be found here:

A shallow copy means constructing a new collection object and then populating it with references to the child objects found in the original. In essence, a shallow copy is only one level deep. The copying process does not recurse and therefore won't create copies of the child objects themselves.

A deep copy makes the copying process recursive. It means first constructing a new collection object and then recursively populating it with

copies of the child objects found in the original. Copying an object this way walks the whole object tree to create a fully independent clone of the original object and all of its children.

Here is an example for the copy():

```
In [104...first_list = [[1, 2, 3], ['a', 'b', 'c']]

    second_list = first_list.copy()

first_list[0][2] = 831

print(first_list)
print(second_list)
```

```
[[1, 2, 831], ['a', 'b', 'c']] [[1, 2, 831],
['a', 'b', 'c']]
```

Here is an example for the deepcopy() case:

```
In [105...import copy

                    first_list = [[1, 2, 3], ['a', 'b', 'c']]

second_list = copy.deepcopy(first_list)

first_list[0][2] = 831

print(first_list)
print(second_list)
```

```
[[1, 2, 831], ['a', 'b', 'c']] [[1, 2, 3],
['a', 'b', 'c']]
```

94. You can avoid throwing errors when trying to access a non-existent key in a dictionary

If you use a usual dictionary and try to access a non-existent key, then you are going to get an error:

```
In [106...my_dictonary = {"name": "Name", "surname": "Surname"}
print(my_dictonary["age"])
```

```
--------------------------------------------------------------------------- KeyError Traceback
(most recent call last) Input In [106], in <cell line: 2>()
1 my_dictonary = {"name": "Name", "surname": "Surname"}
----> 2 print(my_dictonary["age"])


KeyError: 'age'
```

Here it's the error thrown:

KeyError: 'age'

We can avoid such errors using defaultdict():

```
In [107...from collections import defaultdict

my_dictonary = defaultdict(str)
my_dictonary['name'] = "Name"
           my_dictonary['surname'] = "Surname"

            print(my_dictonary["age"])
```

95. You can build your own iterator

```
In [108...class OddNumbers:
def __iter__(self):
                    self.a = 1
return self

def __next__(self):
                    x = self.a
self.a += 2
return x


odd_numbers_object = OddNumbers()
          iterator = iter(odd_numbers_object)

print(next(iterator))
print(next(iterator))
print(next(iterator))
```

```
1
3
5
```

96. You can remove duplicates from a list in a single line

```
In [109...my_set = set([1, 2, 1, 2, 3, 4, 5])
print(list(my_set))
```

```
[1, 2, 3, 4, 5]
```

97. Print the place where a module is located

```
In [110...import torch

  print(torch)
```

```
<module 'torch' from '/home/arjun/.config/jupyterlab-desktop/jlab_server/lib/python3.8/site-packages/torch/__init__.py'>
```

98. You can check whether a value is not part of a list using “not in”

```
In [111...odd_numbers = [1, 3, 5, 7, 9]
even_numbers = []

for i in range(9):
```

```
    if i not in odd_numbers:
        even_numbers.append(i)

print(even_numbers)
```

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

99. Difference between sort() and sorted()

sort() sorts the original list.

sorted() returns a new sorted list.

In [112...

```
groceries = ['milk', 'bread', 'tea']

new_groceries = sorted(groceries)
    # new_groceries = ['bread', 'milk', 'tea']
print(new_groceries)

# groceries = ['milk', 'bread', 'tea']
print(groceries)
groceries.sort()

# groceries = ['bread', 'milk', 'tea']
print(groceries)
```

```
['bread',    'milk',    'tea']
['milk',    'bread',    'tea']
['bread', 'milk', 'tea']
```

100. Generate unique IDs using the uuid module

UUID stands for Universally Unique Identifier.

In [113...

```
import uuid

# Generate a UUID from a host ID, sequence number, and the current time
print(uuid.uuid1()) # 308490b6-afe4-11eb-95f7-0c4de9a0c5af
# Generate a random UUID
print(uuid.uuid4())
```

```
3c2ce9fc-99cd-11ed-bd1b-
d8bbc1acd933    af7edad4-c348-410d-
ac1a-fd1da7989059
```

Bonus: 101. String is a primitive data type in Python

If you come from a Java background, you know that String in Java is a non-primitive data type, because it refers to an object.

In Python, a string is a primitive data type.