



Get unlimited access

Open in app



Published in Geek Culture

You have 3 free member-only stories left this month. [Upgrade for unlimited access.](#)

Chouaieb Nemri

Follow

May 10 · 10 min read ★ · [Listen](#)

Save

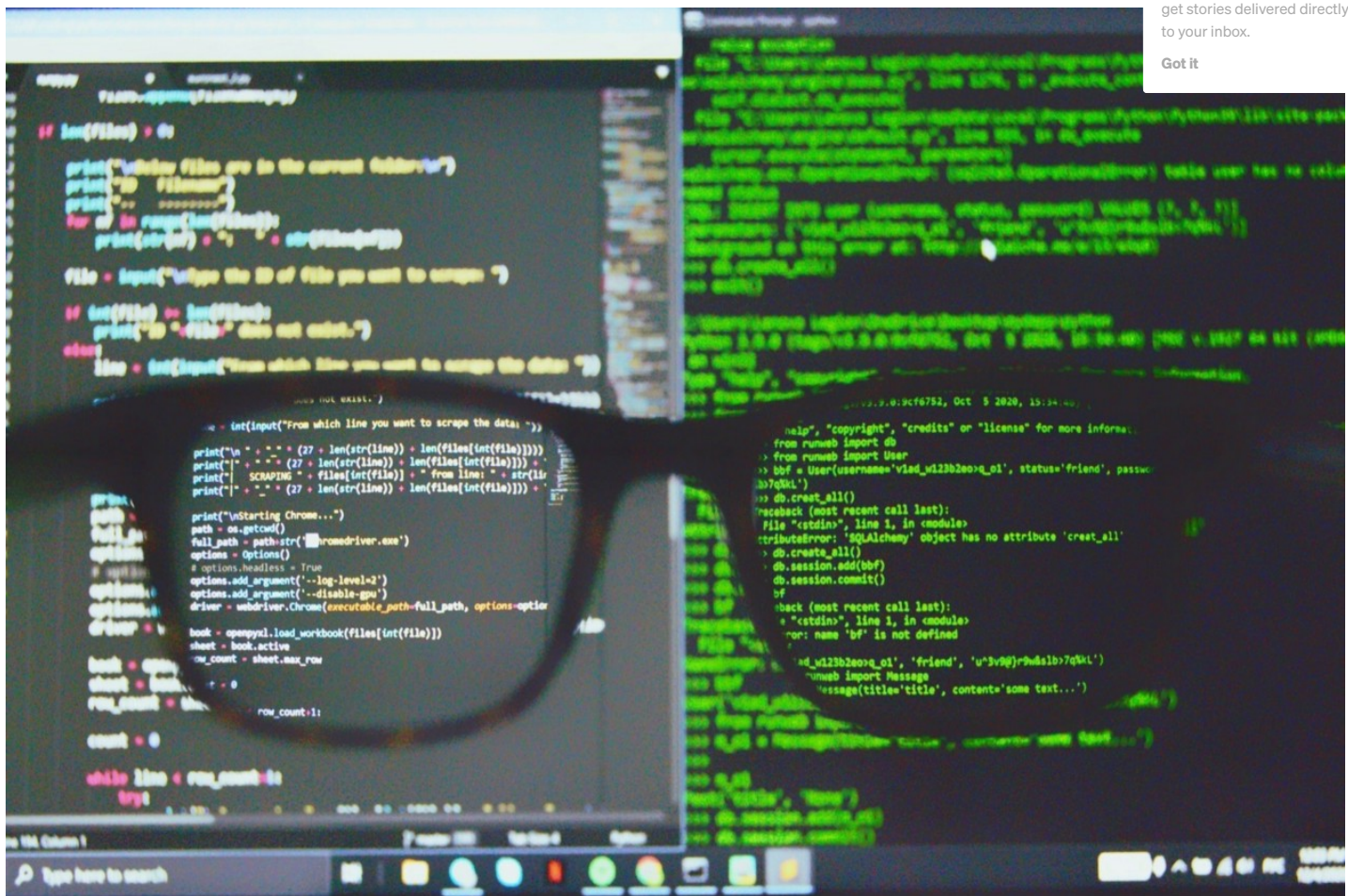


9 Advanced Python Concepts You Should Know About

Note: I have received no compensation for writing this piece. Please consider supporting mine and others' writing by [becoming a Medium member with this link](#).

You can now subscribe to get stories delivered directly to your inbox.

Got it



Python is a general-purpose programming language that is loved for its readability, object-oriented features and its great community support. Apart from being used in web applications, Python is also used in various fields such as data science, artificial intelligence, scientific computing and more. So, if you have been thinking about getting into programming and looking for a general-purpose language, Python may be the right fit for you. In this article, you will learn some advanced Python concepts that will help you get ahead while staying grounded. You don't have to be a seasoned Python programmer to read this article; it is only going to help you understand the language better and make you a better developer. So, let's get into it.

Comprehensions





Get unlimited access

Open in app



Comprehensions are a way to create lists and dictionaries without using loops. They come in three types: list comprehensions, dictionary comprehensions, and set comprehensions. Use comprehensions if you want to create a new list, dictionaries or sets out of an existing iterable. The following code snippet shows you these usages:

```
# Create a list to be used in comprehensions
numbers = [1, 2, 3, -3, -2, -1]

# Create a new list of these numbers' squares
mylist = [x*x for x in numbers] [1, 4, 9, 9, 4, 1]

# Create a new dictionary of these numbers' exponentiation
mydict = {x: pow(10, x) for x in numbers}
# output {1: 10, 2: 100, 3: 1000, -3: 0.001, -2: 0.01, -1: 0.1}

# Create a set of these numbers' absolutes
myset = {abs(x) for x in numbers}
# output {1, 2, 3}
```

These comprehensions have a similar syntax. Here is a brief overview of the different forms follows. It's worth noting that you may set conditions to ensure that you retain the elements you need.

```
List Comprehension: [expr for x in iterable]
Dictionary Comprehension: {key_expr: value_expr for x in iterable}
Set Comprehension: {expr for x in iterable}
With Optional Conditions:
[expr for x in iterable if condition]
{key_expr: value_expr for x in iterable if condition}
{expr for x in iterable if condition}
```

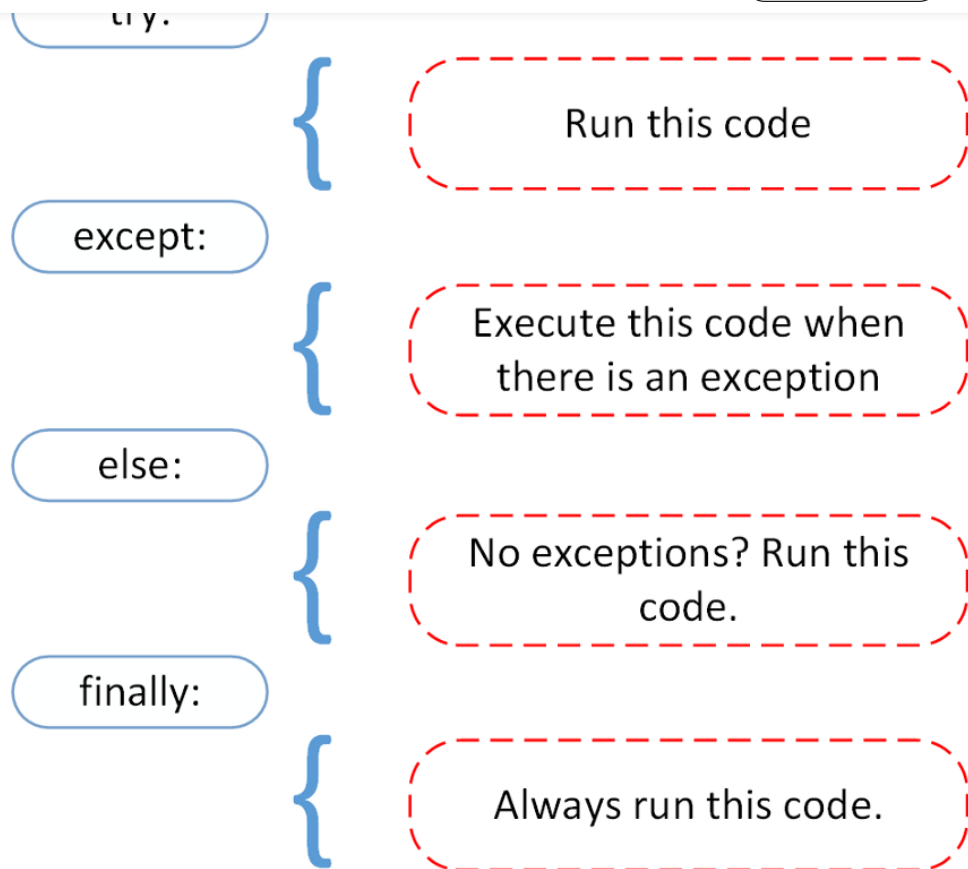
Exception Handling





Get unlimited access

Open in app



An exception is a condition that arises during the program's execution and causes it to be interrupted. It can happen for a variety of reasons. Assume you're building a division program, and the denominator contains 0, resulting in a `ZeroDivisionError`. Importing a library that doesn't exist or accessing an element that isn't in a list index are two further instances. Python comes with about 30 built-in exceptions.

`try` and `except` blocks are used to handle exceptions in python. We can use multiple `except` blocks when we need to handle multiple exceptions at the same time.

`try` blocks has the instructions to be executed. `except` blocks contain the code that executes if `try` fails to execute. We also have an `else` and `finally` blocks. `else` block executes solely when a `try` block is successfully executed. A `finally` block will always execute, no matter the outcome of previous blocks.

Handling a single exception

```
try:
    a = int(input("Enter numerator:"))
    b = int(input("Enter denominator:"))
    c = a / b
    print(c)
except ZeroDivisionError as e:
    print(e, " please provide a non zero denominator")
```

Handling multiple exceptions

```
import sys
try:
    f = open('myfile.txt')
    c = f.readline()
```





Get unlimited access

Open in app

```
except:
    print("Unexpected error:", sys.exc_info()[0])
    raise
finally:
    print("Operation Successfully Done!!") (Example Taken From Official Python Docs)
```

Collections

<code>namedtuple()</code>	factory function for creating tuple subclasses with named fields
<code>deque</code>	list-like container with fast appends and pops on either end
<code>Counter</code>	dict subclass for counting hashable objects
<code>OrderedDict</code>	dict subclass that remembers the order entries were added
<code>defaultdict</code>	dict subclass that calls a factory function to supply missing values

A collection is an object that stores a group of other objects. A list, for example, is a collection of items. Many Python libraries have been created to support additional data structures. The collection library is a Python library designed to enhance the capabilities of the built-in containers. The following are the five most commonly used data structures in the collection module:

1. Counter

Takes an iterable as input and returns a dictionary where the keys are iterable elements and the values are their respective count of occurrence in the original iterable.

```
from collections import Counter
data = [1,1,1,1,2,3,4,3,3,5,6,7,7]
count = Counter(data)
print(count)

-----
Counter({1: 4, 3: 3, 7: 2, 2: 1, 4: 1, 5: 1, 6: 1})
```

2. namedtuple

It is used to assign more meaning to each position of a tuple:

```
from collections import namedtuple
Direction = namedtuple('Direction', 'N,S,E,W')
dt = Direction(4,74,0,0)
print(dt)

-----
Direction(N=4, S=74, E=0, W=0)
```

3. OrderedDict

This is a dictionary structure that remembers the key insertion order. The vanilla dict type incorporates this feature in newer versions of python:

```
from collections import OrderedDict
dictt = OrderedDict()
dictt['a'] = 5
dictt['d'] = 2
dictt['c'] = 1
dictt['b'] = 3
print(dictt)

-----
OrderedDict([('a', 5), ('d', 2), ('c', 1), ('b', 3)])
```

4. defaultdict





Get unlimited access


Open in app

```
dictt['a'] = 2
print(dictt['a']) ## return the value
print(dictt['b']) ## returns the default value
-----
2
0
```

5. deque

deque is a low-level and highly optimized double-ended queue that's useful for implementing elegant, efficient, and Pythonic queues and stacks.

Itertools



```
from itertools import product

if __name__ == "__main__":
    arr1 = ['a', 'b', 'c']
    arr2 = ['d', 'e', 'f']

    print(list(product(arr1, arr2)))

#prints [(a,d),(a,e),(a,f),(b,d),(b,e),(b,f),(c,d),(c,e),(c,f)]
```

The Python itertools module provides various functions that work on iterators.

1. `product(iterable, iterable)` cartesian product of two iterables
2. `permutation(iterable)` all possible permutations with no repeated elements
3. `combinations(iterable, n)` all possible combinations of `n` elements from iterable without replacement
4. `combinations_with_replacement(iterable, n)` all possible combinations of `n` elements from iterable with replacement
5. `accumulate(iterable)` returns the cumulative sum of elements of the iterable
6. `groupby(iterable, key=FUNC)` returns an iterator with consecutive keys and groups from the iterable

Decorators

Decorators are a way in Python to modify the behavior of functions and classes. They allow you to change the functions by adding methods or altering parameters, or classes by adding attributes. A decorators function has an `@` before the function name.

For instance, if you want to log every time the “my_function” function is called you could write:





```
@logging_func
def add(a, b):
    return a + b

result = add(5, 6)
print(result)
```

Let's understand the above decorator's example — first, we have a function name `add` whose work is to take two variables and returns the sum of them. Now after working sometime we realize that there is a need to logging functionality to the same function. Now we have two options the first one is to add the function call logging code in the same `add` function, or we can use decorators to add the functionality without explicitly changing the function.

To use decorators we first defined a decorator function. This function takes `original_func` as input. Then, we have another function. It is a wrapper function that has `*args`, `**kwargs` function parameters. With these, both defined as parameters now we can pass any number of parameters inside the function. In the body of the wrapper function, we have the logic for the logging functionality. When we call the `add` function with some parameters `add(5,6)` the output will be:

```
#output
#Called add with (5, 6) {}
#11
```

Generators

A generator is a function that returns an iterable sequence of values. Unlike the list, which returns all the elements at once and consumes memory for the entire list length, a generator yields items one by one. It contains at least a `yield` statement. `yield` is a keyword in python that is used to return a value from a function without destroying its current state or reference to a local variable. A function with a `yield` keyword is called a generator.

```
----- Fibonacci Series Using Generators -----
def fibon(limit):
    a,b = 0,1
    while a < limit:
        yield a
        a, b = b, a + b
    for x in fibon(10):
        print (x)
```

Dunder Methods

Dunder methods, also known as Magic methods, have two underscores `__` before and after the method name. On a certain action, these methods are called straight from the class. When you use the `*` a sign to multiply two numbers, the internal `__mul__` procedure is invoked.

```
num = 5
num*6
>> 30

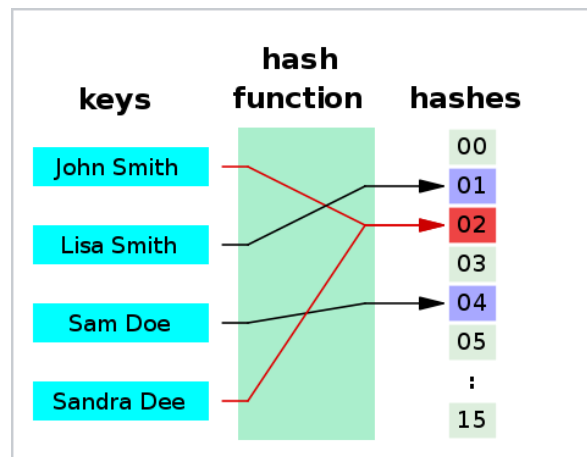
num.__mul__(6)
>>30
```

Typically, these methods are used to overload a predefined operator. For example, the numerical operators `+`, `-`, `*`, `/` must be used around the numerical object, but `+` can also be used to concatenate two strings. As a result, we may argue that the `+` operator is overworked when it comes to performing numerous jobs.

```
5+6
>>11

"python"+"programming"
>> 'pythonprogramming'
```





Hashing is the process of converting Python objects (called keys in the diagram) to numeric hashed values using a hash function (also referred to as hasher) (called hashes in the diagram). Using the built-in `hash()` method to get the hash value of a Python object is a simple way to find out if it exists. Python will throw a `TypeError` exception if the object isn't hashable.

```
# Get an string object's hash value
hash("This is me")
5361907397716593195

# Get a tuple object's hash value
hash((1,2))
-3550055125485641917

# Get a list object's hash value
hash([1, 2, 3])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'

# Get a dict object's hash value
hash({"a": 1, "b": 2})
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'dict'
```

Hashing, in particular, requires time and can be slower than constructing lists and tuples. So, why do we bother creating dictionaries using hashing in the first place? On a similar issue, you may have heard that set items must be hashable as well. Both dictionaries and sets necessitate the creation of hash tables underneath the hood. The following code snippet demonstrates how the hashability of specific objects can influence their suitability for usage as dictionary keys.

The most significant benefit of hashes is that they provide instant look-up time (i.e., $O(1)$ time complexity) when retrieving a dictionary element. Checking whether a specific item is in the set takes the same amount of time. In other words, employing hashing as the implementation mechanism reduces the overhead of having the hash table under the hood while increasing the efficiency of several common operations like item retrieval, item insertion, and item validation.

```
import random
import timeit

# Create a function to check the look up time
def dict_look_up_time(n):
    numbers = list(range(n))
    random.shuffle(numbers)
    d0 = {x: str(x) for x in numbers}
    random_int = random.randint(0, n - 1)
    t0 = timeit.timeit(lambda: d0[random_int], number=10000)
    return t0
```

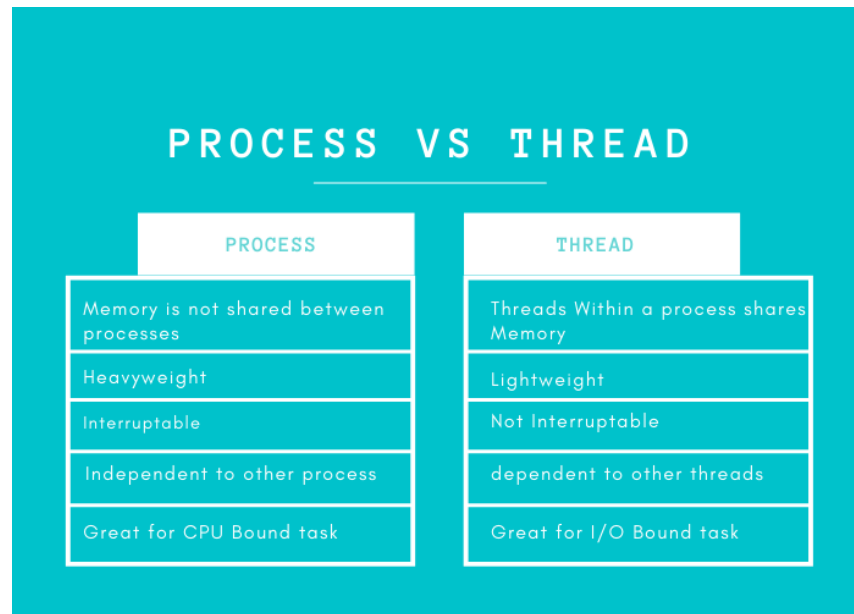




```
*** N = 100 : 0.00256
*** N = 1000 : 0.00291
*** N = 10000 : 0.00207
*** N = 100000 : 0.00286
```

We produce some random integers to determine an average lookup time for item fetching to simulate a real situation. As you can see, even with 100,000 entries in the dictionary, the lookup time is nearly the same, demonstrating the benefit of using a hash table as the storage mechanism for dictionaries.

Threading and Multiprocessing



Threading and multiprocessing are two methods for running many scripts at the same time. A thread is an entity in a process, and a process is an instance of the program.

Multiprocessing is a technique in which many processes operate on different CPUs at the same time. Threading is a technique in which numerous threads run at the same time to complete distinct tasks.

Let's take a look at an example where threading might come in handy. Suppose you have a function which takes time to complete and you need to execute it again and again with different parameters. You can use threading to make this process faster by executing your function in parallel with different parameters. Multiprocessing may come in handy when you want to use the same kind of code for repetitive tasks (for example, processing multiple images). Instead of running this task on a single processor, you can use multiprocessing so that all processors will be used simultaneously.

Next steps? Work on side projects

The best way to become a proficient programmer is by doing projects. Pick something that interests you and work on it for a few weeks. If you're not sure what you want to work on, start with one of these 9 advanced Python concepts and solve the problems that come up. You'll be amazed at how much more you will learn about programming just by following through with a project.

About Me



[Get unlimited access](#)[Open in app](#)

CHOUAIEB NEMRI
DATA - CLOUD - AI

[Let's connect on LinkedIn.](#) I am a Machine Learning and Data Analytics Solutions Architect at AWS (Amazon Web Services). I am passionate about Cloud, Data & AI, I hold Engineering and MS degrees from Georgia Institute of Technology 🇺🇸 and INP Toulouse ENSEEIHT 🇫🇷.

I mentor people with disabilities to land a career in tech: Book your spot with me here 📅 <https://calendly.com/nemri/techies-w-disabilities>

I write about Data, Cloud and AI here 📖 <https://c-nemri.medium.com/>

Sign up for Geek Culture Hits

By Geek Culture

Subscribe to receive top 10 most read stories of Geek Culture — delivered straight into your inbox, once a week. [Take a look.](#)

[Get this newsletter](#)

Emails will be sent to cesarnr21@gmail.com.
[Not you?](#)

