**Exploring Python’s Lesser-Known Standard Libraries**

[[Raman Bazhanau](https://medium.com/@ramanbazhanau?source=post_page-----670c509bf062--------------------------------)](https://medium.com/@ramanbazhanau?source=post_page-----670c509bf062--------------------------------)

[Raman Bazhanau](https://medium.com/@ramanbazhanau?source=post_page-----670c509bf062--------------------------------)

·

[Follow](https://medium.com/m/signin?actionUrl=https%3A%2F%2Fmedium.com%2F_%2Fsubscribe%2Fuser%2Fa082bddc70ad&operation=register&redirect=https%3A%2F%2Fmedium.com%2F%40ramanbazhanau%2Fexploring-pythons-lesser-known-standard-libraries-670c509bf062&user=Raman+Bazhanau&userId=a082bddc70ad&source=post_page-a082bddc70ad----670c509bf062---------------------post_header-----------)

8 min read

·

Sep 23

58

1

Pythons library is a goldmine of modules that cater to various programming requirements. Although modules such, as *sys*, *os*, *datetime*and *json*are well known and frequently used there exist other lesser known modules that provide robust functionalities capable of greatly enriching your Python development journey.



Photo by [Brecht Corbeel](https://unsplash.com/@brechtcorbeel?utm_source=medium&utm_medium=referral) on [Unsplash](https://unsplash.com/?utm_source=medium&utm_medium=referral)

In this article we will delve into some of these hidden treasures: *collections*, *itertools*, *functools*.

**collections**

The collections module provides alternative container datatypes to Python's general-purpose built-ins.

**namedtuple**

A namedtuple is a subclass of a tuple, but with named fields. It's a quick way to create simple classes for storing data without the overhead of defining custom methods.

from collections import namedtuple  
  
# Define a namedtuple  
Person = namedtuple('Person', ['name', 'age', 'gender'])  
  
# Create an instance  
john = Person(name='John Doe', age=30, gender='Male')  
  
print(john.name) # Output: John Doe

Use Cases:

* Storing data records without creating full-fledged classes.
* When you need immutable data structures.

**deque**

A deque (pronounced "deck") stands for "double-ended queue". It supports adding and removing elements from both ends in O(1) time, unlike lists which have O(n) time complexity for append and pop operations from the left.

from collections import deque  
  
d = deque('ghi')  
d.append('j') # add to the right  
d.appendleft('f') # add to the left  
d.pop() # remove from the right  
d.popleft() # remove from the left

Use Cases:

* Implementing queues and stacks.
* Maintaining a list of “last seen items” or a sliding window of items.

**Counter**

A Counter is a dict subclass that counts hashable objects. It's an unordered collection where elements are stored as dictionary keys and their counts as dictionary values.

from collections import Counter  
  
c = Counter('abracadabra')  
print(c) # Output: Counter({'a': 5, 'b': 2, 'r': 2, 'c': 1, 'd': 1})

Use Cases:

* Counting occurrences of items in a list.
* Finding the most common elements in an iterable.

**OrderedDict**

An OrderedDict is a dictionary subclass that remembers the order in which its contents are added. Before Python 3.7, regular dictionaries didn't guarantee order, but this changed in Python 3.7, making OrderedDict less critical.

from collections import OrderedDict  
  
od = OrderedDict()  
od['a'] = 1  
od['b'] = 2  
od['c'] = 3  
print(od) # Output: OrderedDict([('a', 1), ('b', 2), ('c', 3)])

Use Cases:

* When you need to maintain the order of items for logic or output.
* Implementing LRU (Least Recently Used) caches.

**defaultdict**

A defaultdict is a dictionary subclass that provides a default value for a nonexistent key, eliminating the need to check for a key before accessing its value.

from collections import defaultdict  
  
dd = defaultdict(int)  
dd['key\_not\_exist'] += 1  
print(dd) # Output: defaultdict(<class 'int'>, {'key\_not\_exist': 1})

Use Cases:

* Grouping items in a list by a key.
* Counting items without initializing keys.

**ChainMap**

A ChainMap groups multiple dictionaries into a single mapping. Lookups search the dictionaries in the order they are provided.

from collections import ChainMap  
  
dict1 = {'a': 1, 'b': 2}  
dict2 = {'b': 3, 'c': 4}  
combined = ChainMap(dict1, dict2)  
print(combined['b']) # Output: 2

Use Cases:

* Managing multiple configuration settings.
* Overriding default configurations with user-defined values.

**itertools**

Python’s itertools module, part of the standard library, is a collection of tools that allow for more efficient and expressive iteration. It provides a set of fast, memory-efficient tools that are useful in both simple and complex scenarios.

**count**

The count function returns an iterator that produces consecutive integers, indefinitely.

from itertools import count  
  
for i in count(10):  
 if i > 15:  
 break  
 print(i)  
  
# Output:  
# 10  
# 11  
# 12  
# 13  
# 14  
# 15

Use Cases:

* Generating an infinite sequence of numbers.
* Creating a counter.

**cycle**

The cycle function returns an iterator that repeats the elements of an input iterable indefinitely.

from itertools import cycle  
  
counter = 0  
for item in cycle('ABC'):  
 if counter > 5:  
 break  
 print(item)  
 counter += 1  
  
# Output:  
# A  
# B  
# C  
# A  
# B  
# C

Use Cases:

* Repeating a sequence indefinitely.
* Simulating cyclic data structures.

**repeat**

The repeat function returns an iterator that produces the same value each time it's accessed.

from itertools import repeat  
  
for \_ in repeat('A', 3):  
 print(\_)  
  
# Output:  
# A  
# A  
# A

Use Cases:

* Filling a list with a repeated value.
* Generating a constant stream of data.

**chain**

The chain function takes several iterables as arguments and returns a single iterator that produces the contents of all of them as one sequence.

from itertools import chain  
  
for char in chain('ABC', 'DEF'):  
 print(char)  
  
# Output:  
# A  
# B  
# C  
# D  
# E  
# F

from itertools import chain  
  
chained = chain('ABC', 'DEF') # itertools.chain object  
print(\*chained)  
  
# Output:  
# A B C D E F

Use Cases:

* Combining multiple iterables without concatenating them.
* Iterating over several sequences consecutively.

**compress**

The compress function filters one iterable with another.

from itertools import compress  
  
data = ['A', 'B', 'C', 'D']  
selectors = [True, False, True, False]  
  
result = list(compress(data, selectors))  
print(result) # Output: ['A', 'C']

Use Cases:

* Filtering elements based on a separate list of Boolean values.
* Conditional data selection.

**dropwhile**

The dropwhile function drops the elements from an iterable as long as a predicate function returns True, and then returns every element after.

from itertools import dropwhile  
  
for item in dropwhile(lambda x: x < 5, [1, 4, 6, 4, 1]):  
 print(item)  
  
# Output:  
# 6  
# 4  
# 1

Use Cases:

* Filtering out leading items.
* Skipping over initial data that doesn’t meet a condition.

**takewhile**

The takewhile function is the opposite of dropwhile. It returns elements from an iterable as long as the predicate function returns True.

from itertools import takewhile  
  
for item in takewhile(lambda x: x < 5, [1, 4, 6, 4, 1]):  
 print(item)  
  
# Output:  
# 1  
# 4

Use Cases:

* Taking items until a condition is met.
* Extracting a sequence of data without processing the entire iterable.

**permutations**

The permutations function returns all possible orderings of an input.

from itertools import permutations  
  
for p in permutations('AB'):  
 print(p)  
  
# Output:  
# ('A', 'B')  
# ('B', 'A')

Use Cases:

* Generating all possible arrangements of a set of items.
* Solving combinatorial problems.

**combinations**

The combinations function produces all the ways to pick k items from an input iterable.

from itertools import combinations  
  
for combo in combinations('ABC', 2):  
 print(combo)  
  
# Output:  
# ('A', 'B')  
# ('A', 'C')  
# ('B', 'C')

Use Cases:

* Generating all possible selections of a set of items.
* Finding pairs, triplets, etc., in a dataset.

**functools**

Python’s functools module, part of the standard library, provides higher-order functions and operations on callable objects. It's a collection of tools that assist with functional programming tasks, making function manipulation more expressive and efficient.

**lru\_cache**

The lru\_cache decorator caches the results of function calls, so repeated calls with the same arguments can return instantly instead of re-computing results.

from functools import lru\_cache  
  
@lru\_cache(maxsize=None)  
def fib(n):  
 if n < 2:  
 return n  
 return fib(n-1) + fib(n-2)

Use Cases:

* Speeding up expensive function calls.
* Memoization of recursive functions.

**cache**

The @cache decorator automatically caches the results of function calls. This means that subsequent calls with the same arguments can be returned instantly, as the results are stored and retrieved from the cache, eliminating the need for re-computation.

from functools import cache  
  
  
@cache  
def factorial(n):  
 return n \* factorial(n-1) if n else 1

Use Cases:

* Caching results of expensive computations for instant retrieval.
* Simplifying code for memoization, especially for recursive functions.

**cached\_property**

The @cached\_property decorator is used in a class context. It transforms a method into a property whose value is calculated once and stored for subsequent accesses. This is ideal for expensive computations in class instances that, once calculated, do not change for the lifetime of the instance.

from functools import cached\_property  
  
  
class Circle:  
 def \_\_init\_\_(self, radius):  
 self.radius = radius  
  
 @cached\_property  
 def area(self):  
 print("Calculating area...")  
 return 3.14159 \* self.radius \*\* 2

Use Cases:

* Reducing computation in class properties that are expensive to calculate and don’t change.
* Improving performance in object-oriented designs by avoiding repeated calculations.

**partial**

The partial function allows you to "freeze" some portion of a function's arguments and keywords, resulting in a new function with fewer arguments.

from functools import partial  
  
def multiply(x, y):  
 return x \* y  
  
double = partial(multiply, 2)  
print(double(4)) # Output: 8

Use Cases:

* Creating specialized versions of general functions.
* Fixing certain arguments of a function in callbacks.

**reduce**

The reduce function successively applies a binary function to the elements of an iterable, reducing the iterable to a single accumulated result.

from functools import reduce  
  
def add(x, y):  
 return x + y  
  
result = reduce(add, [1, 2, 3, 4, 5])  
print(result) # Output: 15

Use Cases:

* Accumulating results from an iterable.
* Transforming a sequence into a single value.

**wraps**

The wraps decorator is used to update a wrapper function to look more like the wrapped function by copying attributes such as the docstring, name, and module.

from functools import wraps  
  
def my\_decorator(f):  
 @wraps(f)  
 def wrapper(\*args, \*\*kwargs):  
 print("Before calling the function")  
 result = f(\*args, \*\*kwargs)  
 print("After calling the function")  
 return result  
 return wrapper  
  
@my\_decorator  
def say\_hello(name):  
 """Greet someone."""  
 print(f"Hello, {name}!")  
  
print(say\_hello.\_\_name\_\_) # Output: say\_hello  
print(say\_hello.\_\_doc\_\_) # Output: Greet someone.

Use Cases:

* Preserving metadata when creating decorators.
* Ensuring that decorated functions retain their original information.

**singledispatch**

The singledispatch decorator transforms a regular function into a single-dispatch generic function, allowing function overloading based on the type of the first argument.

from functools import singledispatch  
  
@singledispatch  
def fun(arg, verbose=False):  
 if verbose:  
 print("Let me just say,", end=" ")  
 print(arg)  
  
@fun.register(int)  
def \_(arg, verbose=False):  
 if verbose:  
 print("Strength in numbers, eh?", end=" ")  
 print(arg)  
  
@fun.register(list)  
def \_(arg, verbose=False):  
 if verbose:  
 print("Enumerate this:")  
 for i, elem in enumerate(arg):  
 print(i, elem)  
  
fun("Hello, world!")  
fun([1, 2, 3], verbose=True)  
  
# Output:  
# Hello, world!  
# Enumerate this:  
# 0 1  
# 1 2  
# 2 3

Use Cases:

* Creating function overloads based on argument types.
* Implementing generic functions.

**Conclusion**

Python, often celebrated for its “batteries-included” philosophy, truly shines when we delve into its rich standard library. Modules like collections, itertools, and functools exemplify this philosophy, offering a plethora of tools that elevate the programming experience. Whether you're structuring data with specialized containers, iterating over sequences in sophisticated ways, or enhancing the functionality of your functions, these modules have got you covered.

The collections module extends Python's basic set of containers, providing specialized datatypes for various tasks. itertools, on the other hand, offers a suite of tools for constructing and interacting with iterators, making loops more expressive and memory-efficient. Lastly, functools brings to the table a set of utilities that aid in functional programming and function manipulation, allowing for more modular and efficient code.

By diving deep into these modules and understanding their offerings, we not only equip ourselves with powerful tools but also embrace the essence of Pythonic programming. The journey through these modules underscores the importance of exploring the standard library, as it often holds the key to writing cleaner, more efficient, and more Pythonic code. So, the next time you find yourself reaching for an external package or crafting a custom solution, take a moment to explore Python’s built-in modules. There’s a good chance you’ll find just the tool you need, waiting to be discovered.