FUNCTIONTOOLS

The functools module, which is included in Python's standard library, provides essential functionality for working with high-order functions (a function that returns a function or takes another function as an argument ). You can reuse or enhance the utility of your functions or callable objects without having to rewrite them using these capabilities. This simplifies the process of building reusable and maintainable code.

The functools module has 11 functions in the current stable release,They are as follows:

* reduce()
* lru\_cache()
* partial()
* partialmethod()
* singledispatch()
* singledispatchmethod()
* cached\_property()
* total\_ordering()
* update\_wrapper()
* wraps()
* cmp\_to\_key()

Reduce():

A function and an iterable are passed to the reduce(function, sequence) function. It applies the supplied function to all members of the iterable in a cumulative manner from left to right before returning a single value.

Syntax :

reduce(function, sequence[, initial]) -> value

lru cache()

is a decorator that wraps a function in a memoizing callable that saves up to maximize the results of a function call and returns the stored value when the function is called again with the same inputs. When an expensive or I/O bound function is called repeatedly with the same arguments, it can save time.

It primarily employs two data structures: a dictionary to map a function's parameters to its output and a linked list to maintain track of the function's.

Syntax:

lru\_cache(maxsize=128, typed=False)

partial():

Partial functions are derived functions with some input parameters that have already been assigned.The partial() method in Functool is used to construct partial functions/objects, which is an important feature because it allows for:

* Replication of existing functions with some parameters already set.
* In a well-documented manner, create a newer version of an existing function.

The partial function also has valuable characteristics that can be used to track partial functions and objects. These are some of them:

* args - Returns the preassigned positional arguments to the partial function.
* keywords - Returns the pre-assigned keyword arguments to the partial function.
* func - Returns the name of the parent function as well as its location.

Syntax :

partial(func, /, \*args, \*\*keywords)

* using the conventional function

|  |
| --- |
| **def** squared(num):  **return** pow(num, 2)  print(list(map(squared, range(0, 10))))  # Output: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81] |

* using partical() from the functools

**from** functools **import** partial

print(list(map(partial(pow, exp**=**2), range(0, 10))))

# Output: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

### partialmethod():

a method specification rather than a callable method. You can think of it as a method's partial().The partialmethod() method provides a new partial method descriptor, which is similar to partial but is intended to be used as

syntax :

partialmethod(func, \*args, \*\*keywords)

### singledispatch():

The first is a generic function, which is a function made up of numerous functions that all perform the same task for different types. The dispatch algorithm determines which implementation will be utilized during a call.

The second is the Single dispatch, which is a type of generic function dispatch in which the implementation is determined by a single argument's type.

With this in mind, the singledispatch decorator in the functool converts a basic function into a generic function whose behavior is determined by the type of its first parameter. It is used for function overloading in plain English.

### singledispatchmethod():

It's a decorator that works in the same way as @singledispatch, but for methods instead of functions.

### cached\_property():

The cached property() decorator changes a class method into a property whose value is calculated only once and then cached as a normal attribute throughout the life of the instance, as the name suggests. Except for the caching functionality, it's comparable to @property. It's handy for attributes of instances that are normally functionally permanent but are computationally expensive.

### total\_ordering

If you want to make a class that compares different numbers, for example. All of the rich comparison methods would almost certainly need to be implemented. However, because this is likely to be tiresome and unnecessary, you can just implement the \_\_eq\_\_ and \_\_gt\_\_ methods and rely on total ordering to fill in the gaps.

### update\_wrapper():

It makes a wrapper function's metadata look like the wrapped function. In the case of partial functions, update wrapper(partial, parent) will update the partial function's documentation( doc ) and name( name ) to match the parent function's.

Syntax:

update\_wrapper(wrapper, wrapped[, assigned][, updated])

### wraps():

It's just a shortcut for calling update wrapper() on the decorated function. It's the same as calling partial(update wrapper, wrapped=wrapped, assigned=assigned, updated=updated, wrapped=wrapped).

### cmp\_to\_key();

It converts a key function from an old-style comparison function. Any callable that accepts two parameters, compares them and returns a negative number for less-than, zero for equality, or a positive number for greater-than is referred to as a comparison function. The operator.itemgetter() key function is an example of a callable that accepts one argument and returns another value to be used as the sort key. Tools like sorted(), min(), max(), and itertools.groupby use key functions ().

Syntax:

function(iterable, key=cmp\_to\_key(cmp\_function))

Itertools

Itertool is one of the most amazing Python 3 standard libraries. Python provides excellent documentation of the itertools but in this tutorial, we will discuss few important and useful functions or iterators of itertools The key thing about itertools is that the functions of this library are used to make memory-efficient and precise code.

Before learning the Python itertools, you should have knowledge of the Python iterator and generators. In this article, we will describe itertools for beginners are well as for professionals.

“This module implements a number of iterator building blocks inspired by constructs from APL,HASKELL,and SML

In simple words, the number of iterators can together create 'iterator algebra' which makes it possible to complete the complex task. The functions in itertools are used to produce more complex iterators. Let's take an example: [Python built-in zip() function](https://www.javatpoint.com/python-zip-function)

accepts any number of arguments as iterable. It iterates over tuples and return their corresponding elements

* 1. a = [1,2,3]
  2. b= ['a', 'b', 'c']
  3. c = zip(a,b)
  4. print(c)

output:

[(1, 'a'), (2, 'b'), (3, 'c')]

[(1, 'a'), (2, 'b'), (3, 'c')]

[(1,  In the above code, we have passed two lists [1,2,3] and ['a', 'b', 'c'] as iterable in **zip()**function**.**These lists return one element at a time. In [Python](https://www.javatpoint.com/python-tutorial)

, an element that implement **.\_\_iter\_\_()** method called iterable.

The [Python zip() function](https://www.javatpoint.com/python-zip-function)

calls **iter()** on each of its argument and then calls **next()** by combining the result into tuple

Types of Iterator

There are various types of iterator in itertools module. The list is given below:

* Infinite iterators
* Combinatoric iterators
* Terminating iterators

### Infinite Iterators

Any object that can implement **for loop** is called iterators. Lists, tuples, set, dictionaries, strings are the example of iterators but iterator can also be infinite and this type of iterator is called **infinite iterator**

|  |  |  |
| --- | --- | --- |
| **Iterator** | **Argument** | **Results** |
| count(start,step) | start, [step] | start, start+step, step+2\*step |
| cycle() | P | p0,p1,….plast |
| repeat() | elem [,n] | elem, elem, elem,….endlessly or upto n times |

**1.count(start, stop)**: It prints from the start value to infinite. The step argument is optional, if the value is provided to the**step** then the number of steps will be skipped.

**import** itertools

**for** i in itertools.count(10,5):

**if** i == 50:

**break**

**else**:

print(i,end=" ")

output:

10 15 20 25 30 35 40 45

**2.cycle(iterable)**: This iterator prints all value in sequence from the passed argument. It prints the values in a cyclic manner. Consider the following example:

**import** itertools

temp = 0

**for** i in itertools.cycle("123"):

**if** temp > 7:

**break**

**else**:

        print(i,end=' ')

        temp = temp+1

output:

1 2 3 1 2 3 1 2 3 1 2

**3.repeat(val,num)**: As the name suggests, it repeatedly prints the passed value for infinite time. The **num**argument is option.

Combinatoric iterator

The complex combinatorial constructs are simplified by the recursive generators. The permutations, combinations, and Cartesian products are the example of the combinatoric construct.

In Python, there are four types of combinatoric iterators:

1. **Product() -**It is used to calculate the cartesian product of input iterable. In this function, we use the optional **repeat** keyword argument for computation of the product of an iterable with itself. The **repeat** keyword represents the number of repetitions. It returns output in the form of sorted tuples
2. **Permutations()**: It is used to generate all possible permutation of an iterable. The uniqueness of each element depends upon their position instead of values. It accepts two argument **iterable** and **group\_size**. If the value of group\_size is **none** or not specified then group\_size turns into length of the iterable.
3. **Combinations()**: It is used to print all the possible combinations (without replacement) of the container which is passed as argument in the specified group size in sorted order
4. **Combination\_with\_replacement()**: It accepts two arguments, first argument is a r-length tuple and the second argument is repetition. It returns a subsequence of length n from the elements of the iterable and repeat the same process. Separate elements may repeat itself in **combination\_with\_replacement()**

**Terminating iterator**

erminating iterators are generally used to work on the small input sequence and generate the output based on the functionality of the method used in iterator.

There are different types of terminating iterator:

1. **accumulate(iter, func)**: It takes two arguments, the first argument is iterable and the second is a function which would be followed at each iteration of value in iterable. If the function is not defined in **accumulate()** iterator, addition takes place by default. The output iterable depends on the input iterable; if input iterable contains no value then the output iterable will also be empty.
2. **dropwhile(func, seq)** - It starts printing the character only after the **func**.
3. **filterfalse(func,seq)** - We can assume it by its name, as this iterator prints only those values that return false for the passed function.
4. **islice(iterable,start,stop,step)** - It slices the given iterable according to given position. It accepts four arguments respectively and these are iterable, container, starting pos., ending position and step(optional).
5. **starmap(func, tuple list)** - It takes two arguments; first argument is function and second argument is list which consists element in the form of tuple.
6. **takewhile(func, iterable)** - It is visa-versa of **dropwhile().** It will print values until it returns false condition.
7. **tee(iterator, count)** - It divides the container into a number of iterators which is defined in the argument.
8. **zip\_longest(iterable1, iterable2, fillval)** - It prints the values of iterable alternatively in sequence. If one of the iterable prints all values, remaining values are filled by the values assigned to fill value

**TQDM**

Tqdm got its name from the Arabic name taqaddum which means 'progress'. tqdm is a library in Python which is used for creating Progress Meters or Progress Bars. tqdm got its name from the Arabic name ***taqaddum***which means ‘progress’. Implementing tqdm can be done effortlessly in our loops, functions or even Pandas. bars are pretty useful Progress in Python because:

Using TQDM

tqdm gives a console kind of progress bar for our processes. Using tqdm is a straightforward process explained in the following steps

First, we need to install our required library tqdm. Open a New Jupyter Notebook and execute

This will complete the installation of tqdm. Restart the Kernel to start using the tqd

!pip install time

2.import the libraries :

Import the newly installed tqdm and time libr\*-ary

From tqdm import tqdm

3.using tqdm :

Now we will use the function tqdm() on a simple program with a  *for loop*.

Ex : For i in tq (range(20) ):

Time.sleep (0.5)

Here **i** is the variable that takes a value of the number 0 to 19 during each iteration. During each iteration, the system will sleep for 0.5 seconds before moving to the next iteration.

The complete code would look like this:

On **Completion**of Code Execution, we get:

From tqdm import tqdm

Import time

For i in tq(range(20)):

Time.sleep(0.5)

We can also give attributes to tqdm() such as desc, which takes a string and will get added as a prefix before the progress bar. Thus,

From tqdm import tqdm

Import time

For i in tqdm (range(20),desc = ‘tqdm() progress bar’);

Time.sleep(0.5)

Apart from the progress bar, tqdm gives additional information such as the number of iterations completed out of the total number of iterations, Total Elapsed Time, Estimated Time to Complete the whole loop, and the speed of the loop in iterations per second (or it/s).on **Completion**of Code Execution

Using tqdm\_notebook() in pands : **tqdm()** and **tqdm\_notebook()**can be used in Pandas. One way is to use them with **for loops** with Pandas Series which works the same as with the loops we have seen earlier. Another way is to use them in Pandas **.apply()** method use **tqdm()** or **tqdm\_notebook()** for .apply(), the .apply() needs to be replaced by **.progress\_apply().**