A HITCHHIKER'S GUIDE TO `functools`

EuroPython 2021

Scott Irwin

Bloomberg

Engineering

https://sjirwin.github.io/hitchhikers-guide-to-functools/

WHAT WE WILL COVER

- Definitions and history
- Simplifying function signatures
- Function wrappers
- Caching
- Ordered types
- Reduce
- Function overloading
- Conclusion and references

DEFINITIONS AND HISTORY

DEFINITIONS

- The Python Standard Library module `functools` contains several higher-order functions
- **Higher-order functions** are functions which act on or return other functions
    - A function decorator is an example of a higher-order function

HISTORY

- The `functools` module was added in Python 2.5 (2006)
  - `wraps`, `update_wrapper` and `partial`
- Additional functions continue to be added
  - Python 3.0: `reduce` (backported to 2.6)
  - Python 3.2: `total_ordering` and `cmp_to_key` (backported to 2.7) plus `lru_cache`
  - Python 3.4: `partialmethod`, `singledispatch`
  - Python 3.8: `cached_property`, `singledispatchmethod`
  - Python 3.9: `cache`

SIMPLIFYING FUNCTION SIGNATURES

- `partial`
- `partialmethod`

`partial`

- `partial(func, *args, **keywords)`
- Takes as input a function and the arguments to "lock in"
- Returns a `partial object` which behaves like the original function called with those arguments already defined

```
from functools import partial
pow_2 = partial(pow, exp=2)
print(f'{pow_2(5)=}') # pow_2(5)=25
```

REDUCE ARGUMENTS

- Use `partial` to transform a multi-argument function to a single argument function in places where that is required (e.g., `map`)
- Possible without `partial`, but is more verbose

```
>>> from functools import partial
>>> list(map(partial(pow, exp=3), range(10)))
[0, 1, 8, 27, 64, 125, 216, 343, 512, 729]
```

```
>>> def pow_3(x):
...     return pow(x, 3)
>>> list(map(pow_3, range(10)))
[0, 1, 8, 27, 64, 125, 216, 343, 512, 729]

>>> list(map(lambda x: pow(x, 3), range(10)))
[0, 1, 8, 27, 64, 125, 216, 343, 512, 729]
```

SIMPLIFY CODE

- Define functions that are easier to type, read, and friendly to code completion

```
import sys
from functools import partial
print_stderr = partial(print, file=sys.stderr)
print_stderr('This output goes to stderr')
```

`partialmethod`

- `partialmethod(func, *args, **keywords)`
- Easiest to think of it as `partial` for methods
- From the Python docs

> *Returns a new `partialmethod` descriptor which behaves like `partial` except that it is designed to be used as a method definition rather than being directly callable.*

- The function argument must be a descriptor or a callable

EXAMPLE: `partialmethod`

```
# Example code from Python3.9 docs
from functools import partialmethod
class Cell:
    def __init__(self):
        self._alive = False
    @property
    def alive(self):
        return self._alive
    def set_state(self, state):
        self._alive = bool(state)
    set_alive = partialmethod(set_state, True)
    set_dead = partialmethod(set_state, False)
```

```
>>> from cell import Cell
>>> c = Cell()
>>> c.alive
False
>>> c.set_alive()
>>> c.alive
True
```

FUNCTION WRAPPERS

- `wraps`
- `update_wrapper`

`wraps`

- `wraps(wrapped)`
- A function decorator used when defining a wrapper function
- Updates the `wrapper` function attributes to be the same as the `wrapped` function
- Convenience decorator factory defined using `update_wrapper()`

EXAMPLE: WITHOUT `wraps`

```
>>> def my_decorator(f):
...     def wrapper(*args, **kwargs):
...         '''wrapper doc string'''
...         print('wrapper called')
...         return f(*args, **kwargs)
...     return wrapper
>>> @my_decorator
... def func():
...     '''func doc string'''
...     print('func called')
```

```
>>> func()
wrapper called
func called
>>> func.__name__
'wrapper'
>>> func.__doc__
'wrapper doc string'
```

EXAMPLE: USING `wraps`

```
>>> from functools import wraps
>>> def my_decorator_wraps(f):
...     @wraps(f)
...     def wrapper(*args, **kwargs):
...         '''wrapper doc string'''
...         print('wrapper called')
...         return f(*args, **kwargs)
...     return wrapper
>>> @my_decorator_wraps
... def func():
...     '''func doc string'''
...     print('func called')
```

```
>>> func()
wrapper called
func called
>>> func.__name__
'func'
>>> func.__doc__
'func doc string'
```

update_wrapper

- `update_wrapper(wrapper, wrapped)`
- Rarely used directly
- Updates the `wrapper` function attributes to be the same as the `wrapped` function
- Useful in situations where `@wraps` cannot be used
  - Wrapping a function after it is defined
  - Wrapping a function you do not own

EXAMPLE: update_wrapper

```
>>> import string
>>> def my_capwords(s, sep=None):
...     '''my_capwords docstring'''
...     print('my_capwords')
...     return string.capwords(s, sep=sep)
>>> my_capwords('spam spam spam')
my_capwords
'Spam Spam Spam'
>>> my_capwords.__name__
'my_capwords'
```

```
>>> from functools import update_wrapper
>>> capwords = update_wrapper(my_capwords, string.capwords)
>>> string.capwords.__name__, capwords.__name__, my_capwords.__name__
('capwords', 'capwords', 'capwords')
>>> capwords('spam spam spam')
my_capwords
'Spam Spam Spam'
```

CACHING

- `lru_cache`
- `cache`
- `cached_property`

`lru_cache`

- `lru_cache(maxsize=128, typed=False)`
- Wraps a function with a memoizing callable
- Saves time when an expensive function is sometimes called with the same arguments
- Caches results of most recent *maxsize* calls
  - **LRU** stands for **L**east **R**ecently **U**sed
- If *typed* is set to `True`, function arguments of different types will be cached separately

`lru_cache` ATTRIBUTES

- Function is also instrumented with 3 functions
  - `cache_info()` - returns a named tuple showing *hits*, *misses*, *maxsize* and *currsize*
  - `cache_clear()` - clears/invalidates the cache
  - `cache_parameters()` - new `dict` showing the values for *maxsize* and *typed*

EXAMPLE: `lru_cache`

```
from functools import lru_cache
@lru_cache
def fib(n):
    if n < 2:
        return n
    return fib(n-1) + fib(n-2)
```

```
>>> from fib import fib
>>> [fib(n) for n in range(16)]
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610]
>>> fib.cache_info()
CacheInfo(hits=28, misses=16, maxsize=128, currsize=16)
>>> fib.cache_parameters()
{'maxsize': 128, 'typed': False}
```

`lru_cache` CAVEATS

- Function's positional and keyword arguments must be hashable
  - Underlying storage is a dictionary
- Should only be used with pure functions
  - Same inputs always produce the same output
  - No side-effects

`cache`

- Simple lightweight unbounded function cache
- Same as `lru_cache(maxsize=None)`
- Because there is no eviction, it is smaller and faster than `lru_cache()` with a size limit

`cached_property`

- Similar to `property()` with the addition of caching
- Value is computed once and then cached as a normal attribute for the life of the instance
- Unlike `property()`, `cached_property` allows writes without a setter being defined
- `cached_property` only runs on lookup and only if the attribute does not already exist
- Once attribute exists, subsequent reads and writes work like a normal attribute

EXAMPLE: `cached_property`

```
import statistics
from functools import cached_property
class DataSet:
    def __init__(self, sequence_of_numbers):
        self._data = tuple(sequence_of_numbers)
    @cached_property
    def stdev(self):
        return statistics.stdev(self._data)
```

```
>>> from random import random
>>> seq = (random() for _ in range(10_000_000))
>>> from dataset import DataSet
>>> d = DataSet(seq)
>>> import time
>>> time.time(), d.stdev, time.time()
(1626747711.129694, 0.28863495535352907, 1626747739.0964322)
>>> time.time(), d.stdev, time.time()
(1626747753.31688, 0.28863495535352907, 1626747753.3168828)
```

ORDERED TYPES

- `total_ordering`

`total_ordering`

- Class decorator that makes it easy to create well behaved totally ordered types
- If class defines at least one rich comparison operator, it supplies the rest
  - Class must define one of `__lt__()`, `__le__()`, `__gt__()`, or `__ge__()`
  - Additionally, class should supply an `__eq__()` method
- **Caveat:** does come at the cost of slower execution and more complex stack traces for the derived comparison methods

EXAMPLE: total_ordering

```
from functools import total_ordering
@total_ordering
class Car():
    def __init__(self, year, make, model):
        self.year, self.make, self.model = year, make, model
    def __eq__(self, o):
        if not isinstance(o, Car):
            return NotImplemented
        return ((self.year, self.make, self.model) == (o.year, o.make, o.mo
    def __lt__(self, o):
        if not isinstance(o, Car):
            return NotImplemented
        return ((self.year, self.make, self.model) < (o.year, o.make, o.mod
```

```
>>> from car import Car
>>> car_1 = Car(2020, 'BMW', '530i')
>>> car_2 = Car(2020, 'BMW', '330i')
>>> (car_1 < car_2), (car_1 > car_2)
(False, True)
```

REDUCE

- `reduce`

reduce

- `reduce(function, iterable[, initializer])`
- Applies *function* of 2 arguments cumulatively to the items of *iterable* to reduce it to a single value
  - Built-in function sum() is an example of a reducer
- Example:

```
>>> from functools import reduce
>>> import operator
>>> def product(iterable):
...     return reduce(operator.mul, iterable, 1)
>>> product([2,5,8])
80
>>> product(range(2,5))
24
```

FUNCTION OVERLOADING

- `singledispatch`
- `singledispatchmethod`

`singledispatch`

- Function decorator which transforms a function into a single-dispatch generic function
  - Means implementation is chosen based on the type of a single argument
- Generic function is decorated with `@singledispatch`
- Overloaded implementations are decorated with the `register()` attribute of the generic function
  - If implementation is annotated with types, the decorator will automatically infer the type of the argument
  - Otherwise, the type is an argument to the decorator

EXAMPLE: `singledispatch`

```
from functools import singledispatch
@singledispatch
def fun(arg):
    print(f"Let me just say, {arg}")
@fun.register
def _(arg: int):
    print(f"Strength in numbers, eh? {arg}")
@fun.register
def _(arg: list):
    print("Enumerate this:")
    for i, elem in enumerate(arg):
        print(i, elem)
@fun.register(complex)
def _(arg):
    print(f"Better than complicated. {arg.real} {arg.imag}")
```

EXAMPLE: `singledispatch`

```
>>> from fun import fun
>>> fun(9)
Strength in numbers, eh? 9
>>> fun([9,7])
Enumerate this:
0 9
1 7
>>> fun(3.4)
Let me just say, 3.4
>>> fun(3.4 + 6j)
Better than complicated. 3.4 6.0
```

`singledispatchmethod`

- Single dispatch for methods
- Function decorator which transforms a method into a single-dispatch generic function
  - Dispatch happens on the type of the first non-`self` or non-`cls` argument

CONCLUSION

- The `functools` module contains many higher-order functions which are both useful and powerful
- Using these functions can result in more readable and maintainable code

REFERENCES

- Python Language Reference
  - https://docs.python.org/3/library/functools.html
- These slides on GitHub
  - https://sjirwin.github.io/hitchhikers-guide-to-functools/
- Martin Heinz's blog
  Functools - The Power of Higher-Order Functions in Python
  - https://martinheinz.dev/blog/52
- Florian Dahlitz's blog
  Introduction To Python's Functools Module
  - https://florian-dahlitz.de/articles/introduction-to-pythons-functools-module