

Closures, Decorators and Generators



Functional Programming and Stateful Iteration

- Functional Programming
 - Higher order functions
 - Closures
- Decorators
 - Decorator factories
 - Class decorators
- The iteration protocol
- Generators

Functional Programming

- Programming style characterized by
 - Functions
 - No side-effects/mutable-state
 - Higher order functions

Higher Order Functions

- Functions as objects
- Essential features
 - Functions can take functions as inputs
 - Functions can return functions
- Python supports both

Functions as Input

- Consider these two functions:

```
def sum_of_squares(nums):  
    total = 0.0  
    for num in nums:  
        total += num ** 2  
    return total
```

```
def sum_of_cubes(nums):  
    total = 0.0  
    for num in nums:  
        total += num ** 3  
    return total
```

only difference



- There's a lot of duplicated code (one line differs)

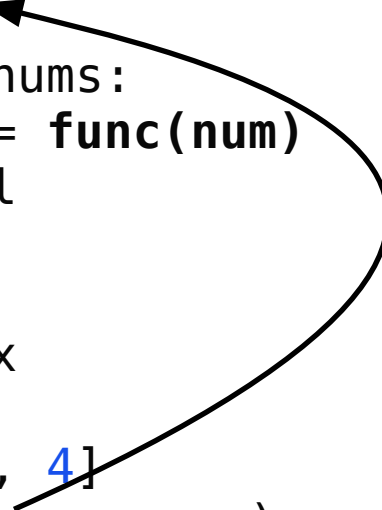
Functions as Input

- Recognizing commonality is part of abstraction

```
def sum_map(func, nums):  
    total = 0.0  
    for num in nums:  
        total += func(num)  
    return total
```

```
def square(x):  
    return x * x
```

```
nums = [1, 2, 3, 4]  
r = sum_map(square, nums)
```



- This version allows any operation to be applied by passing a function
- Mapping an operation across a sequence and reducing to a single value is an example of "map-reduce"

Lambda Functions

- Single expression functions can be written as a lambda

```
def sum_map(func, nums):  
    total = 0.0  
    for num in nums:  
        total += func(num)  
    return total
```

```
nums = [1, 2, 3, 4]  
r = sum_map(lambda x: x * x, nums)
```

- lambda creates an anonymous function
- Can only contain a single expression
- So no control flow or exception handling, etc.
- Commonly used for callback functions/event handlers

Partial Application

- lambda can be used to alter function arguments

```
def distance(x, y):  
    return abs(x - y)
```

```
>>> dist_from10 = lambda y: distance(10, y)  
>>> dist_from10(3)  
7  
>>> dist_from10(14)  
4
```

- functools.partial

```
from functools import partial  
dist_from10 = partial(distance, 10)
```


Returning Functions

- Consider the following function

```
def greet(name):  
    def do_greet():  
        print(f'Hello {name}')    return do_greet
```

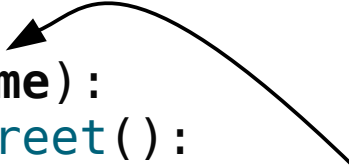
```
>>> greeter = greet('Sally')  
>>> greeter  
<function greet.<locals>.do_greet at 0x7fe99b6c18b0>  
>>> greeter()  
Hello Sally
```

- A function that returns a function!
- Notice that it works, but ponder it...

Nested Scopes


- Observe how the inner function refers to a variable defined by the outer function

```
def greet(name):  
    def do_greet():  
        print(f'Hello {name}')    return do_greet
```



- Even after greet has completed, and the function returned, the name variable is kept alive

```
>>> greeter = greet('Sally')  
>>> greeter()  
Hello Sally
```



Where does the name
variable come from?

Closures

- If a function returns an inner function the result is called a "closure"

```
def greet(name):  
    def do_greet():  
        print(f'Hello {name}')    return do_greet
```

- Essential feature: A "closure" retains the values of all variables needed for the function to run properly later on
- The closure "closes over" any variables it uses from the enclosing scope

Closures

- To make it work references to variables from the outer scope (bound variables) are carried along with the function

```
def greet(name):  
    def do_greet():  
        print(f'Hello {name}')    return do_greet  
  
>>> greeter = greet('James')  
>>> greeter.__closure__  
(<cell at 0x7fe99c4556d0: str object at 0x7fe99b675770>,)  
>>> greeter.__closure__[0].cell_contents  
'James'
```

Note: these are implementation details you'll rarely need to use, but useful for understanding Python.

Closures

- Closures only capture (keep-alive) variables they use

```
def add(x, y):  
    result = x + y  
    def get_result():  
        return result  
    return get_result
```

- The function add has three local variables: x, y, result
- The inner function only uses result

```
>>> a = add(3, 10)  
>>> a.__closure__  
(<cell at 0x7fe99b6cc8b0: int object at 0x956fc0>,)  
>>> a.__closure__[0].cell_contents  
13
```

Closures and Mutability

- Assignment inside a function creates a local variable
- But inner functions can change the value of a variable from an enclosing scope with `nonlocal`

```
def counter(n):  
    def incr():  
        nonlocal n  
        n += 1  
        return n  
    return incr
```

```
>>> c = counter(10)  
>>> c()  
11  
>>> c()  
12
```

Note: functions can be nested to any level; you can have functions within functions within functions. Functions can also return multiple closures.

- Can be used to hold mutable internal state, like a class, or communicate between closures with shared state

Using Closures

- Closures are an essential feature of Python
- Closures encapsulate state
- Create utility functions/tools specialised on the input to the closure. Factory functions.
- Common applications:
 - Delayed evaluation
 - Callback functions
 - Partial application
 - Code creation (metaprogramming)

Decorators

- A decorator is (typically) a function that wraps another function
- With some special Python syntax to enable them
- The wrapper is a new function that works the same way as the old one (same arguments, same return value) except that some kind of extra processing is carried out
- Decorators are built on closures with an inner function
- Let's see a simple example first

Wrapper Functions

- Here is a simple function:

```
def add(x, y):  
    return x + y
```

- Here is an example of a wrapper function:

```
def logged_add(x, y):  
    print(f'Adding {x} + {y}')  
    return add(x, y)
```

- Example use:

```
>>> add(3, 4)
```


```
7
```

```
>>> logged_add(3, 4)
```

```
Adding 3 + 4
```

```
7
```

The extra output is caused by the wrapper, but the original function is still called to get the result.



Creating Wrappers

- Insight: You can write a function that makes a wrapper around any function

```
def logged(func):  
    # Define a wrapper function around func  
    def wrapper(*args, **kwargs):  
        print('Calling', func.__name__)  
        return func(*args, **kwargs)  
    return wrapper
```

- Usage:

```
>>> logged_add = logged(add)  
>>> logged_add  
<function logged.<locals>.wrapper at 0x7fe99b6c18b0>  
>>> logged_add(7, 2)  
Calling add  
9
```

Wrappers as Replacements

- When you create a wrapper, you often want to replace the original function with it

```
def add(x, y):  
    return x + y
```

```
# Replace add with a wrapped version  
add = logged(add)
```

- Other code continues to use the original function name, but can be unaware that it has been replaced by a wrapped version

```
>>> add(3, 4)  
Calling add  
7
```

Decorator Concept

- When you replace a function with a wrapper you usually giving the function extra functionality
- This process is known as decoration
- You are "decorating" a function with extra features
- The extra features can be before calling the original or after calling the original, they can process the arguments to the function or work with the return value or even do function registration and return the function unchanged or tagged

Decorator Syntax

- The definition of a function and wrapping almost always occur together

```
def add(x, y):  
    return x + y  
add = logged(add)
```

- However it looks a little weird (defining something and replacing it immediately) and can be error prone
- The @ syntax simplifies it

```
@logged  
def add(x, y):  
    return x + y
```

Decorator Syntax

- Whenever you see a decorator, a function is being wrapped. That's it
- Example in classes with the builtin decorators:

```
class Foo:
```

```
    @staticmethod  
    def bar():  
        ...
```

```
    @classmethod  
    def spam(cls):  
        ...
```

```
    @property  
    def name(self):  
        ...
```



```
class Foo:
```

```
    def bar():  
        ...  
    bar = staticmethod(bar)  
  
    def spam(cls):  
        ...  
    spam = classmethod(spam)  
  
    def name(self):  
        ...  
    name = property(name)
```

Using Decorators

- Use a decorator every time you want to define a kind of "macro" involving function definitions
- There are many kinds of applications
 - Avoiding code duplication
 - Enabling and disabling optional features
 - Debugging and diagnostics
 - Optimization
 - Function registration

Timing Measurements

- A decorator that reports execution time

```
import time
def timethis(func):
    def wrapper(*args, **kwargs):
        start = time.time()
        r = func(*args, **kwargs)
        end = time.time()
        print(func.__name__, end - start)
        return r
    return wrapper
```

- Usage:

```
@timethis
def bigcalculation():
    statements
    statements
```


Advanced Decorators

- There are a few additional tricky details
 - Multiple decorators
 - Decorator metadata
 - Decorators with arguments

Multiple Decorators

- You can apply as many decorators as you want

```
@foo
@bar
@spam
def add(x,y):
    return x + y
```

- This is the same as this:

```
add = foo(bar(spam(add)))
```

- The decorators are applied "bottom up" (innermost wrapper first)
- Particularly important for the builtin decorators (property, classmethod, etc) which are "descriptors" not functions and must be last (at the top)

Function Metadata

- When you define a function there is metadata (information) attached to the function (the name, the docstring, function signature, function attributes, etc)

```
def add(x,y):  
    "Adds x and y"  
    return x + y
```

```
>>> import inspect  
>>> add.__name__  
'add'  
>>> add.__doc__  
'Adds x and y'  
>>> inspect.signature(add)  
<Signature (x, y)>
```

- This metadata is used by help, IDEs and documentation building tools like Sphinx

The Metadata Problem

- The way we've written decorators so far loses this metadata

```
@logged
def add(x,y):
    "Adds x and y"
    return x + y

>>> add.__name__
'wrapper'
>>> print(add.__doc__)
None
>>> inspect.signature(add)
<Signature (*args, **kwargs)>
```


- This is a problem

Copying Metadata

- Decorators should copy metadata to the new function

```
def logged(func):  
    def wrapper(*args, **kwargs):  
        print('Calling', func.__name__)  
        return func(*args, **kwargs)  
    wrapper.__name__ = func.__name__  
    wrapper.__doc__ = func.__doc__  
    return wrapper
```


Manually copying metadata



- A better solution: use functools.wraps:

```
from functools import wraps  
  
def logged(func):  
    @wraps(func)  
    def wrapper(*args, **kwargs):  
        print('Calling', func.__name__)  
        return func(*args, **kwargs)  
    return wrapper
```

Copies metadata for you



Decorators with Args

- Decorators can take arguments

```
@decorator(x, y, z)
def func():
    ...
```

- These are "decorator factories"
- The decorator factory is called, with arguments, to create the actual decorator
- Its the equivalent of each of these two pieces of code:

```
actual_decorator = decorator(x, y, z)
@actual_decorator
def func():
    ...
```

```
def func():
    ...
func = decorator(x, y, z)(func)
```

Decorators with Args

- Example: logging with a custom message

```
def logmsg(message):  
    def logged(func):  
        @wraps(func)  
        def wrapper(*args, **kwargs):  
            print(message.format(name=func.__name__))  
            return func(*args, **kwargs)  
        return wrapper  
    return logged
```

- Example use:

```
@logmsg('You called {name}')
```

```
def add(x, y):  
    return x + y
```

Decorators with Args

- Example: logging with a custom message

Outer function takes the arguments

```
def logmsg(message):  
    def logged(func):  
        @wraps(func)  
        def wrapper(*args, **kwargs):  
            print(message.format(name=func.__name__))  
            return func(*args, **kwargs)  
        return wrapper  
    return logged
```

Arguments can be used by the code inside

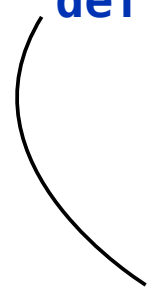
- The outer function is like an enclosing environment that gets added to the decorator to accept the extra arguments
- Two levels of nested scopes!

Decorators with Args

- Example: logging with a custom message

The same decorator as before

```
def logmsg(message):  
    def logged(func):  
        @wraps(func)  
        def wrapper(*args, **kwargs):  
            print(message.format(name=func.__name__))  
            return func(*args, **kwargs)  
        return wrapper  
    return logged
```



- The inner function is a standard decorator
- It uses the message argument from the enclosing scope, the decorator factory

Decorators with Args

- Decorator factories are more general
- You can call them to create decorators which can be used as normal

```
logged = logmsg('Calling {name}')
```

```
@logged  
def add(x, y):  
    return x + y
```

- This is subtle but can simplify code where you have several similar decorators

Class Decorators

- Decorators can also be applied to class definitions

```
@decorator
class Foo:
    def bar(self):
        ...
    def spam(self):
        ...
```

- Exactly the same as doing:

```
class Foo:
    def bar(self):
        ...
    def spam(self):
        ...
```

```
Foo = decorator(Foo)
```

- Manipulates or wraps the class definition

Class Decorators

- Unlike function decorators most class decorators don't wrap the class but inspect or modify it
- Typical prototype:

```
def decorator(cls):  
    # Do something with cls  
    ...  
    # Return the original class back  
    return cls
```

- Notice that here the class is not replaced but the original class is returned

Example

- Recording all attribute lookups

```
def logged_getattr(cls):  
    # Get the original implementation  
    orig_getattribute = cls.__getattribute__  
    # Replacement method  
    def __getattribute__(self, name):  
        print('Getting:', name)  
        return orig_getattribute(self, name)  
    # Attach to the class  
    cls.__getattribute__ = __getattribute__  
    return cls
```

- This is replacing a method on a class

Example

```
@logged_getattr
class Spam:
    def foo(self):
        pass
    def bar(self):
        pass
```

```
>>> s = Spam()
```

```
>>> s.x = 23
```

```
>>> s.x
```

```
Getting: x
```


```
>>> s.foo()
```

```
Getting: foo
```

```
>>> s.bar()
```

```
Getting: bar
```

Notice that all
attribute lookups are
now logged



Decoration via Inheritance

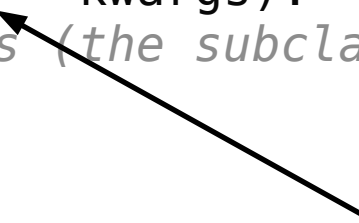
- Base classes can customise subclasses with the `__init_subclass__` method
- Can also be used for class registration, a framework tool

```
class Parent:
    def __init_subclass__(cls, **kwargs):
        # Do something with cls (the subclass)
        ...
```

```
class Child1(Parent):
    pass
```

```
class Child2(Parent, a=1, b=2):
    pass
```

The parent receives the child class as the first argument, `cls`, along with any keyword arguments in the child definition



- Like an implicit/inherited class decorator
- Class decorators and `__init_subclass__` replace most use-cases for metaclasses

Iteration

- Iteration defined: Looping over items

```
a = [2, 4, 10, 37, 62]  
# Iterate over a  
for x in a:  
    ...
```

- A very common pattern, fundamental to computer science
- Loops, list comprehensions, tuple unpacking, equality comparisons with sequences, etc
- Most programs do a huge amount of iteration

Iteration: Protocol

- Iteration

```
for x in obj:  
    # statements  
    ...
```

- Underneath the covers

```
_iter = obj.__iter__()           # Get iterator object  
while True:  
    try:  
        x = _iter.__next__()     # Get next item  
    except StopIteration:        # No more items  
        break  
    # statements  
    ...
```

- Objects that work with the for-loop all implement this low level protocol

Iteration: Protocol

- Example: Manual iteration over a list

```
>>> x = [1, 2, 3]
>>> it = iter(x)      # equivalent of x.__iter__()
>>> next(it)          # equivalent of it.__next__()
1
>>> next(it)
2
>>> next(it)
3
>>> next(it)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Delegating Iteration

- Iteration is part of the container protocol
- The easiest way to make a custom container iterable is to delegate to a builtin iterator

```
class Portfolio:
    def __init__(self):
        self._holdings = []
    def __iter__(self):
        return iter(self._holdings)
```

- An example of object composition with delegation

Generators

- Generators provide "stateful iteration"
- Generators simplify custom iteration, using the yield keyword

```
def countdown(n):  
    print('Counting down from', n)  
    while n > 0:  
        yield n  
        n -= 1
```

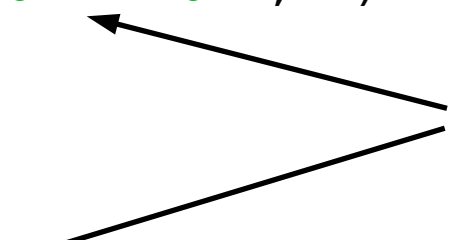
```
>>> for i in countdown(5):  
...     print('T-minus', i)  
...  
Counting down from 5  
T-minus 5  
T-minus 4  
T-minus 3  
T-minus 2  
T-minus 1
```

Generator Functions

- Generators are full coroutines, this is a subset of their functionality
- Behaviour is very different from normal functions
- Calling a generator function creates a generator object (an iterator), it does not start running the function

```
def countdown(n):  
    print('Counting down from', n)  
    while n > 0:  
        yield n  
        n -= 1
```

```
>>> x = countdown(10)  
>>> x  
<generator object at 0x58490>
```

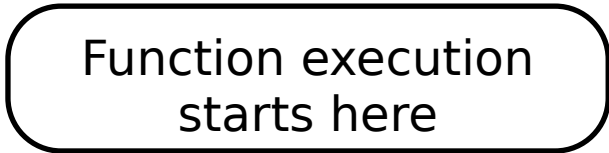


Notice that no
output was
produced!

Generator Functions

- Execution only begins on next (when iteration starts)

```
>>> x = countdown(10)
>>> x
<generator object at 0x58490>
>>> next(x)                                     # invokes x.__next__()
Counting down from 10
10
```



A rounded rectangle with the text "Function execution starts here" has an arrow pointing to the output "Counting down from 10" of the `next(x)` call.

- yield produces a value and pauses execution
- Control flow is yielded along with the value
- When next is called again execution resumes, until we hit yield again

```
>>> next(x)
9
>>> next(x)
8
```

Generator Functions

- When the generator returns, iteration stops

```
>>> next(x)
1
>>> next(x)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

- Observation: A generator function implements the same low level iteration protocol that the for-loop uses on all iterables; lists, tuples, dicts, files, etc
- Any function that contains the "yield" keyword is a generator function

Reusing Generators

- Generators are "one shot", single use

```
>>> c = countdown(5)
>>> for x in c:
...     print('T-minus', x)
...
Counting down from 5
T-minus 5
T-minus 4
T-minus 3
T-minus 2
T-minus 1
>>> for x in c:
...     print('T-minus', x)
...
>>>
```

- To reuse, recreate the generator

```
>>> c = countdown(5)
```


Reusable Generators

- Subtle trick: Making iterable objects with `__iter__`

```
class Countdown:
    def __init__(self, n):
        self.n = n
    def __iter__(self):
        n = self.n
        while n > 0:
            yield n
            n -= 1
```

- The object can be iterated over many times, because every call to `__iter__` produces a new generator