Core Python: Functions and Functional Programming

FUNCTIONS AND CALLABLES

Austin Bingham
COFOUNDER - SIXTY NORTH

@austin_bingham

Robert Smallshire
COFOUNDER - SIXTY NORTH

@robsmallshire

Prerequisites

Function Types

Free functions

Functions defined at module scope.

Methods

Functions defined within a class definition.

Argument Types

Positional arguments are matched with formal arguments by position, in order.

Keyword arguments are matched with formal arguments by name.

The choice between the two is made at the call site.

Default Arguments

Arguments may have a default value .

The default value for an argument is only evaluated once .

Be careful when using mutable data types for default values.

Functions are objects and can be passed around just like any other object.

Function Definition and Invocation

```
>>> import socket
>>> def resolve(host):
...     return socket.gethostbyname(host)
...
>>> resolve
<function resolve at 0x10b412f70>
>>> resolve('sixty-north.com')
'93.93.131.30'
>>>
```

Prerequisites

Core Python: Getting Started

You will need a working Python 3 system.

This material will work in all recent versions of Python 3.

If possible, use the latest stable version.

At a minimum you need need a working Python 3 REPL, though you can use an IDE.

Naming Special Functions

__feature__

Hard to prounounce!

dunder

Our way of pronouncing special names

A portmanteau of "double underscore"

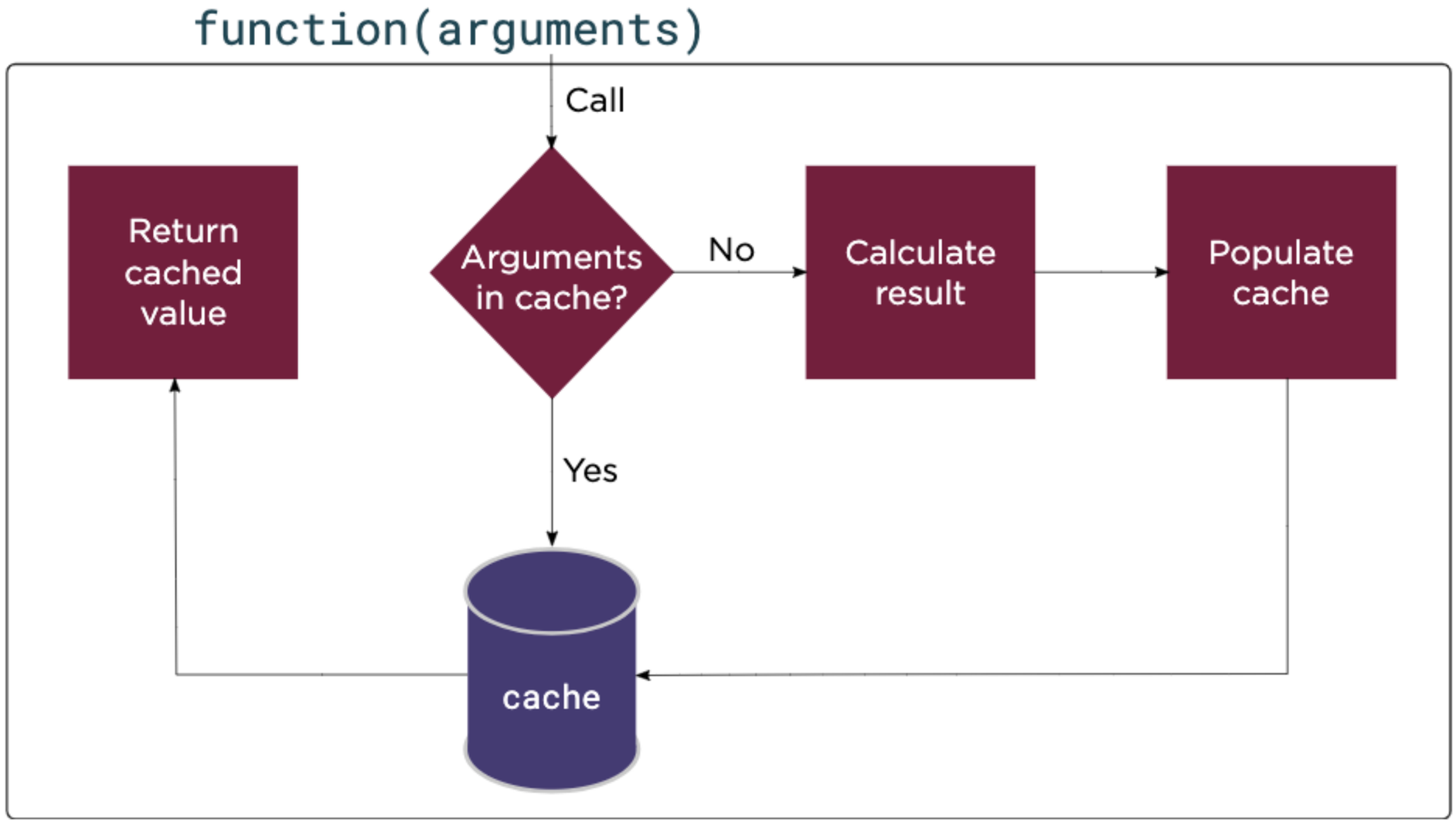
Instead of "underscore underscore len underscore underscore" we'll say "dunder len"

Companion Book Series

leanpub.com/python-journeyman/c/pluralsight

Callable Instances

Caching Function Results



`__call__()`

Allows instances of classes to be callable objects.

`__call()__` is invoked on objects when they are called like functions.

Caching Resolver

```
import socket


class Resolver:
    def __init__(self):
        self._cache = {}

    def __call__(self, host):
        if host not in self._cache:
            self._cache[host] = socket.gethostbyname(host)
        return self._cache[host]
```

Callable Instances

```
>>> from resolver import Resolver
>>> resolve = Resolver()
>>> resolve('sixty-north.com')
'93.93.131.30'
>>> resolve.__call__('sixty-north.com')
'93.93.131.30'
>>> resolve._cache
{'sixty-north.com': '93.93.131.30'}
>>> resolve('pluralsight.com')
'35.163.3.54'
>>> resolve._cache
{'sixty-north.com': '93.93.131.30', 'pluralsight.com': '35.163.3.54'}
>>> from timeit import timeit
>>> timeit(setup="from __main__ import resolve", stmt="resolve('google.com')", n
umber=1)
0.022649178999999853
>>> timeit(setup="from __main__ import resolve", stmt="resolve('google.com')", n
umber=1)
3.9330000000122533e-06
>>> print("{:f}".format(_))
0.000004
>>>
```

Since callable instances are just normal class instances, their classes can define any other methods you want.

Callable Instance with Methods

```
import socket

class Resolver:
    def __init__(self):
        self._cache = {}

    def __call__(self, host):
        if host not in self._cache:
            self._cache[host] = socket.gethostbyname(host)
        return self._cache[host]

    def clear(self):
        self._cache.clear()

    def has_host(self, host):
        return host in self._cache
```

Callable Instance with Methods

```
>>> from resolver import Resolver
>>> resolve = Resolver()
>>> resolve.has_host("pluralsight.com")
False
>>> resolve("pluralsight.com")
'52.38.90.254'
>>> resolve.has_host("pluralsight.com")
True
>>> resolve.clear()
>>> resolve.has_host("pluralsight.com")
False
>>>
```

Classes Are Callable

Class objects and instances of classes are very different things.

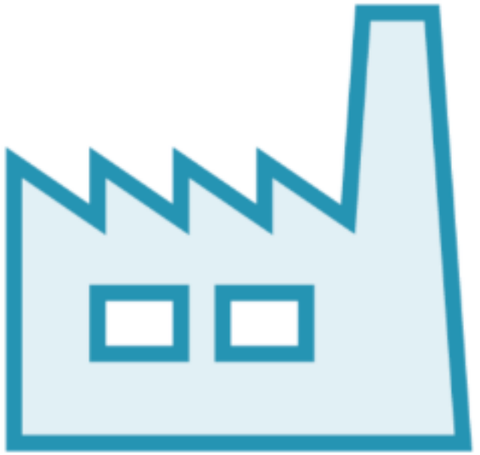
`class` binds a class object to a named reference.

Classes as Callables

```
>>> from resolver import Resolver
>>> Resolver
<class 'resolver.Resolver'>
>>> resolve = Resolver()
>>>
```

Arguments passed to the class object are forwarded to the class's `__init__()`.

Classes Are Object Factories



Classes produce new instances when they are invoked.

Instance construction is covered in later courses.

Returning Class Objects

```
>>> def sequence_class(immutable):
...     if immutable:
...         cls = tuple
...     else:
...         cls = list
...     return cls
...
>>> seq = sequence_class(immutable=True)
>>> t = seq("Timbuktu")
>>> t
('T', 'i', 'm', 'b', 'u', 'k', 't', 'u')
>>> type(t)
<class 'tuple'>
>>>
```

Alternatives to Class

`cls`

Shortened version "class"

Very common in the Python ecosystem

klass

Deliberate misspelling of "class"

A bit more explicit

Conditional expressions

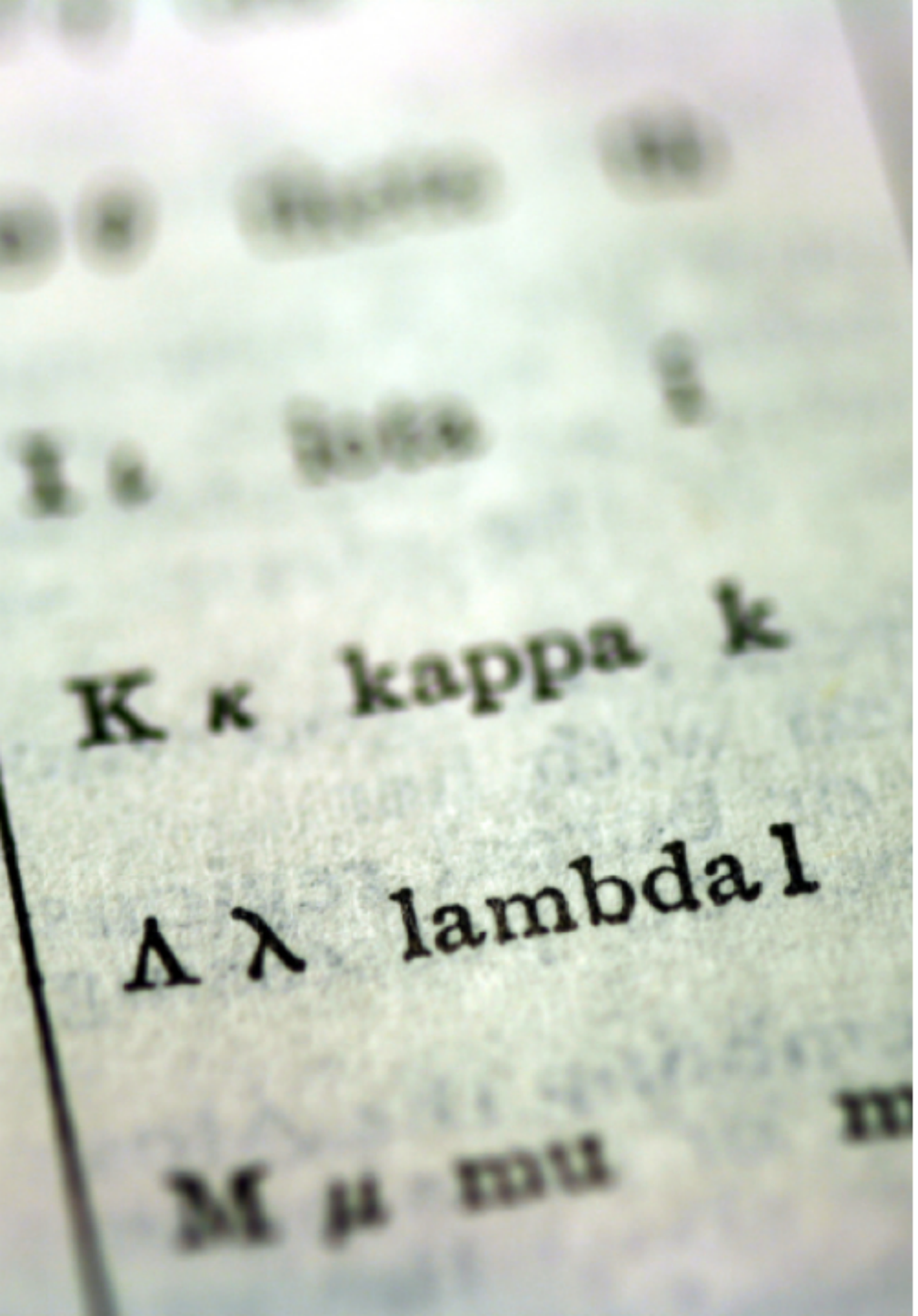
Evaluates to one of two expressions depending on a boolean.

```
result = true_value if condition else false_value
```

Conditional Expressions

```
>>> def sequence_class(immutable):
...     return tuple if immutable else list
...
>>> seq = sequence_class(immutable=False)
>>> s = seq("Nairobi")
>>> s
['N', 'a', 'i', 'r', 'o', 'b', 'i']
>>> type(s)
<class 'list'>
>>>
```

Lambdas



In many cases anonymous callable objects will suffice.

`lambda` allows you to create such anonymous callable objects.

Use `lambda` with care to avoid creating inscrutable code.

Why do we use the Greek letter lambda?

It's due to Alonzo Church's work on the foundations of computer science in 1936.

His lambda calculus forms the basis for many modern functional languages.

Effective Use of Lambdas

```
sorted(iterable,key)
```

list of names

lambda

Sorting with a Lambda

```
>>> scientists = ['Marie Curie', 'Albert Einstein', 'Rosalind Franklin',
...               'Niels Bohr', 'Dian Fossey', 'Isaac Newton',
...               'Grace Hopper', 'Charles Darwin', 'Lise Meitner']
>>>
>>> sorted(scientists, key=lambda name: name.split()[-1])
['Niels Bohr', 'Marie Curie', 'Charles Darwin', 'Albert Einstein', 'Dian Fossey'
, 'Rosalind Franklin', 'Grace Hopper', 'Lise Meitner', 'Isaac Newton']
>>> last_name = lambda name: name.split()[-1]
>>> last_name
<function <lambda> at 0x10e630f70>
>>> last_name("Nikola Tesla")
'Tesla'
>>> def first_name(name):
...     return name.split()[0]
...
>>>
```

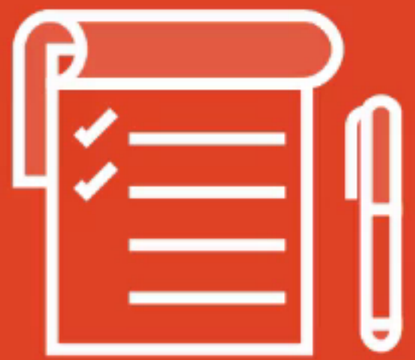
Functions vs. Lambdas

`def name(args): body`	`lambda args: expr`
Statement which defines a function and binds it to a name	Expression which evaluates to a function
Must have a name	Anonymous
Arguments delimited by parentheses, separated by commas	Argument list terminated by a colon, separated by commas
Zero or more arguments supported - zero arguments ⇒ empty parentheses	Zero or more arguments supported - zero arguments ⇒ `lambda:`
Body is an indented block of statements	Body is a single expression
A `return` statement is required to return anything other than `None`	The return value is given by the body expression; no `return` statement is permitted
Regular functions can have docstrings	Lambdas cannot have docstrings
Easy to access for testing	Awkward or impossible to test

Detecting Callable Objects

```
>>> def is_even(x):
...     return x % 2 == 0
...
>>> callable(is_even)
True
>>> is_odd = lambda x: x % 2 == 1
>>> callable(is_odd)
True
>>> callable(list)
True
>>> callable(list.append)
True
>>> class CallMe:
...     def __call__(self):
...         print("Called!")
...
>>> my_call_me = CallMe()
>>> callable(my_call_me)
True
>>> callable("This is not callable")
False
>>>
```

Summary



Reviewed basics of Python functions

Use `__call__()` to make callable instances

Associating state with callable objects

Classes are callable objects

Calling a class object creates an instance of the class

Summary

Lambdas are unnamed callable objects

When to use lambdas vs. functions and other callables

Use `callable()` to determine if an object is callable

Conditional expressions are a concise form of conditionals

Classes are objects