VTI Academy

WAY TO ENTERPRISE

Basic Programming

Lesson 08

VTI Academy

Functional Programming

Higher-order Functions in Python

A higher-order function is one that takes a function as a parameter or returns a function as a result, or both.

```python
def f(x):
    return x + 2

def g(h, x):
    return h(x) * 2

print(g(f, 42))
```

Composition

```python
def addx(x):
    def _(y):
        return x + y
    return _

add2 = addx(2)
add3 = addx(3)

print(add2(2), add3(3))
```

Closure

```python
def f(x, y):
    return x*y

def f2(x):
    def _(y):
        return f(x, y)
    return _

print(f2(2))
print(f2(2)(3))
```

Currying

One Function or Three?

```python
def get_ints(ints, odd=True, even=True):
    if odd and even:
        return [i for i in ints]
    elif odd:
        return [i for i in ints if i % 2]
    elif even:
        return [i for i in ints if not i % 2]
    else:
        return []
```

Do it all function

```python
def get_even_ints(ints):
    return [i for i in ints if not i % 2]

def get_odd_ints(ints):
    return [i for i in ints if i % 2]

def get_all_ints(ints):
    return list(ints)
```

Split them!

Trouble in Mutable Town

```
pi = 3.14159

def change_pi(pi):
    pi = 2.71828

print(pi)
change_pi(pi)
print(pi)
```

Try to change pi

```
pi = 3.14159

def change_pi(*args, **kwargs):
    global pi
    pi = 2.71828

print(pi)
change_pi(pi)
print(pi)
```

Really change pi!

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'>
>>>
```

VTI Academy

Lambdas

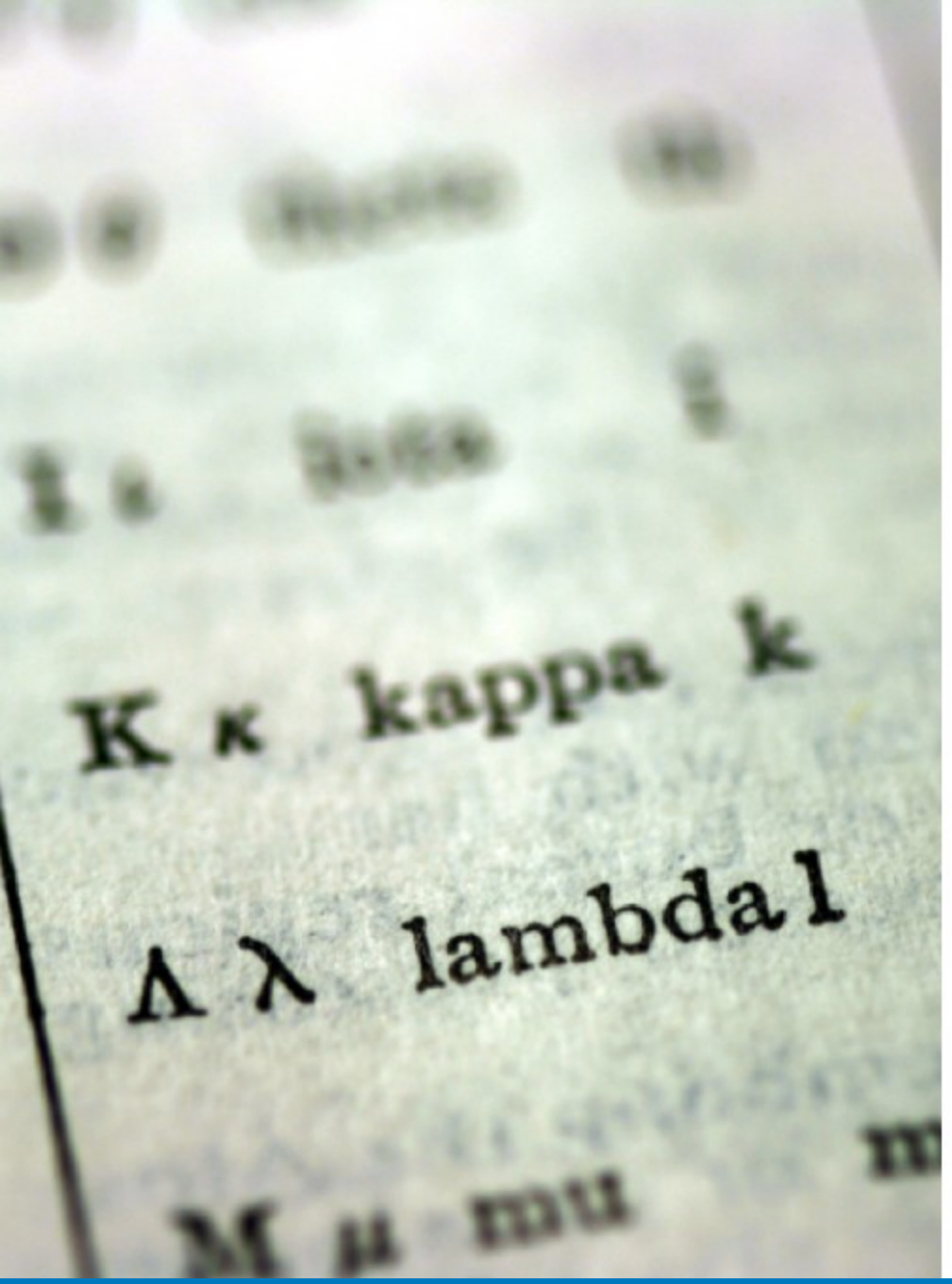
```
def g(h, x):
    return h(x) * 2

print(g(lambda x: x + 2, 42))
```

Lambdas in Python

Import the dis module

Define a simple test function

Disassemble it

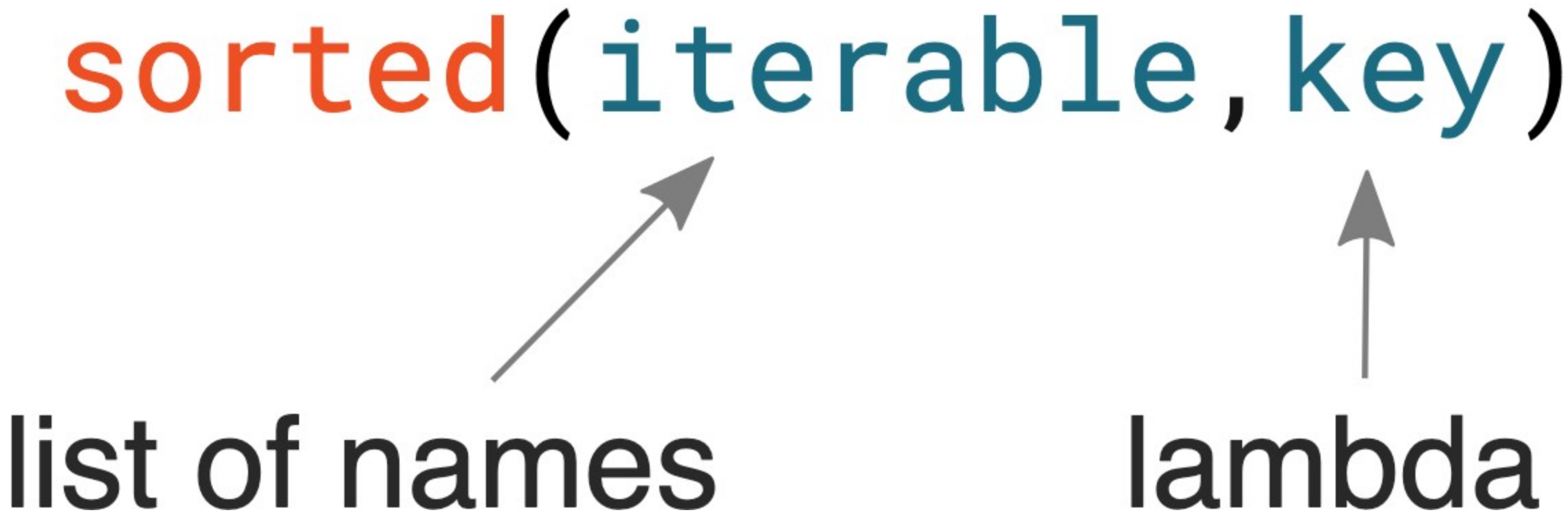


In many cases anonymous callable objects will suffice.

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

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

Effective Use of Lambdas



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

Extended Argument Syntax

Extended Argument Syntax

```
>>> print()

>>> print("one")
one
>>> print("one", "two")
one two
>>> print("one", "two", "three")
one two three
>>> "{a}<===>{b}".format(a="Oslo", b="Stavanger")
'Oslo<===>Stavanger'
>>>
```

Hypervolume

```
>>> hypervolume(3, 4, 5)
(3, 4, 5)
<class 'tuple'>
>>> def hypervolume(*lengths):
...     i = iter(lengths)
...     v = next(i)
...     for length in i:
...         v *= length
...     return v
...
>>> hypervolume(2, 4)
8
>>> hypervolume(2, 4, 6)
48
>>> hypervolume(2, 4, 6, 8)
384
>>> hypervolume(1)
1
>>> hypervolume()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in hypervolume
StopIteration
>>>
```

Extended Call Syntax

Extended Call Syntax

```
>>> def print_args(arg1, arg2, *args):
...     print(arg1)
...     print(arg2)
...     print(args)
...
>>> t = (11, 12, 13, 14)
>>> print_args(*t)
11
12
(13, 14)
>>>
```

Extended Call Syntax for Mappings

```
>>> def color(red, green, blue, **kwargs):
...     print("r =", red)
...     print("g =", green)
...     print("b =", blue)
...     print(kwargs)
...
>>> k = {'red':21, 'green':68, 'blue':120, 'alpha':52 }
>>> color(**k)
r = 21
g = 68
b = 120
{'alpha': 52}
>>> k = dict(red=21, green=68, blue=120, alpha=52)
>>>
```

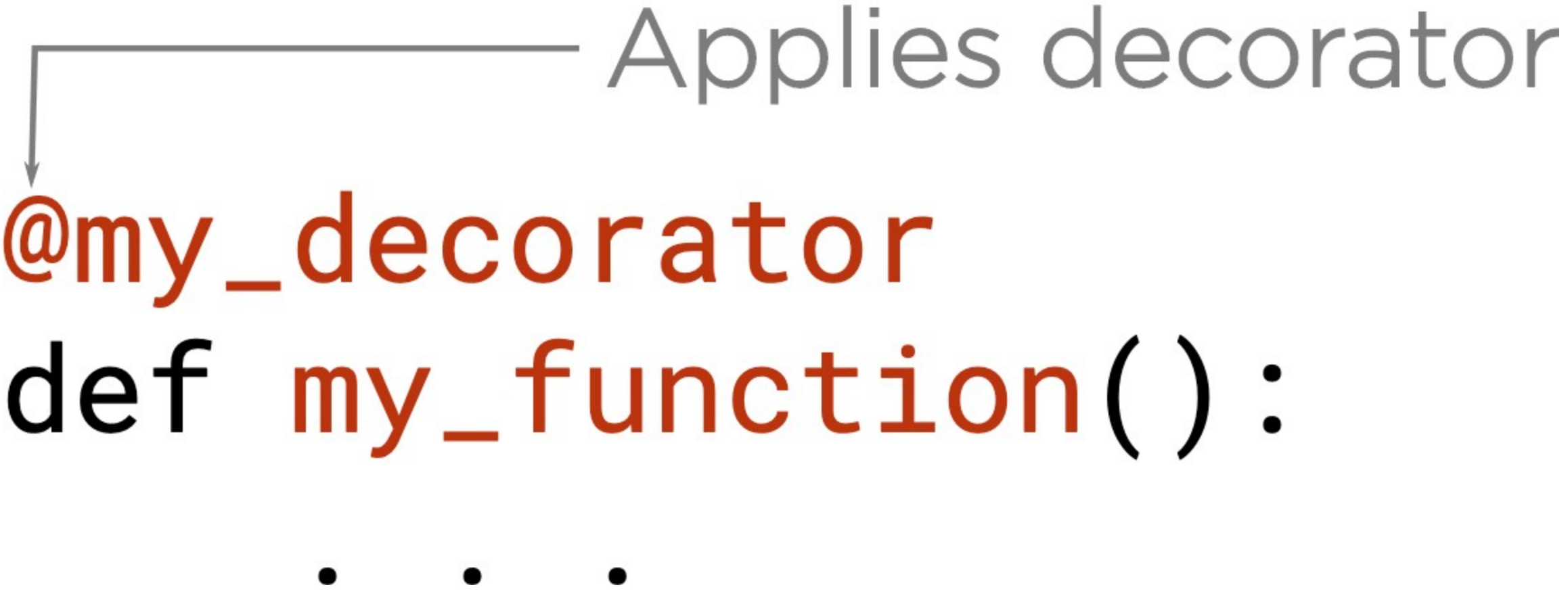
Argument Forwarding

```
>>> def trace(f, *args, **kwargs):
...     print("args =", args)
...     print("kwargs =", kwargs)
...     result = f(*args, **kwargs)
...     print("result =", result)
...     return result
...
>>> trace(int, "ff", base=16)
args = ('ff',)
kwargs = {'base': 16}
result = 255
255
>>>
```

VTI Academy

Function Decorators

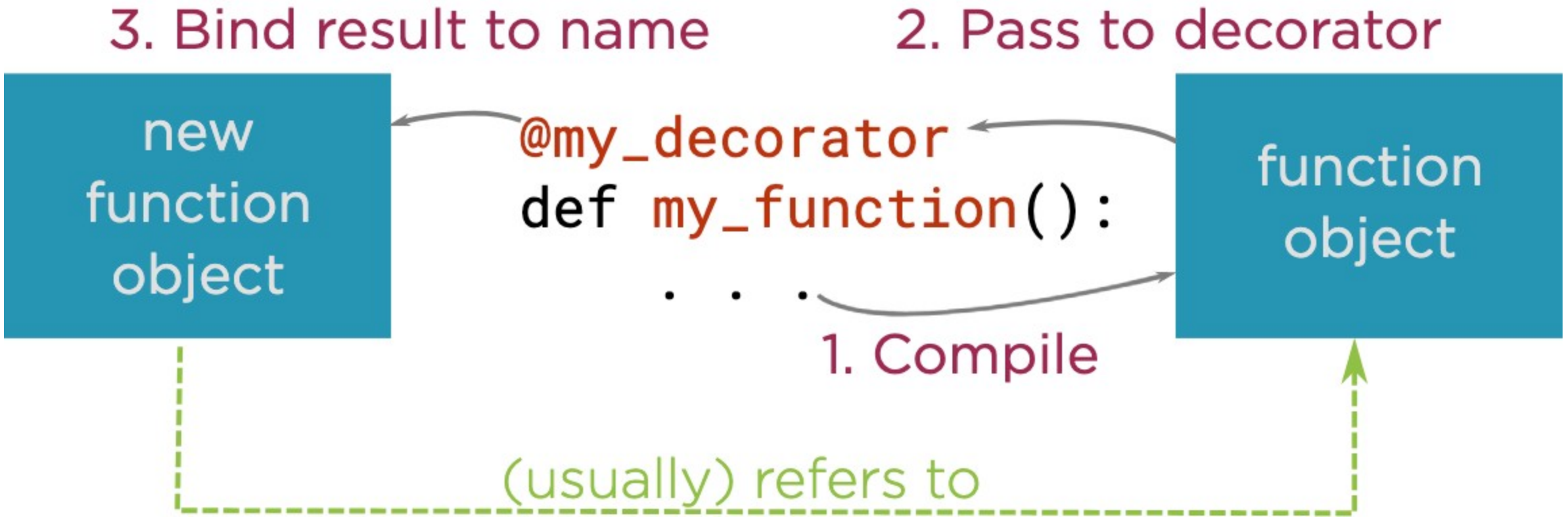
Decorator Syntax



VTI Academy

Decorators allow you to modify existing functions without changing their definition.
Callers don't need to change when decorators are applied.

Decorator Application



Decorator Example

```
>>> def escape_unicode(f):
...     def wrap(*args, **kwargs):
...         x = f(*args, **kwargs)
...         return ascii(x)
...     return wrap
...
>>> def northern_city():
...     return 'Tromsø'
...
>>> print(northern_city())
Tromsø
>>> @escape_unicode
... def northern_city():
...     return 'Tromsø'
...
>>> print(northern_city())
'Troms\xf8'
>>>
```

Classes as Decorators

Classes as Decorators

1. Classes are **callable objects**
2. Functions decorated with a class are replaced by an **instance of the class**
3. These instances **must themselves be callable**

We can decorate with a class as long as instances of the class implement `__call__()`.

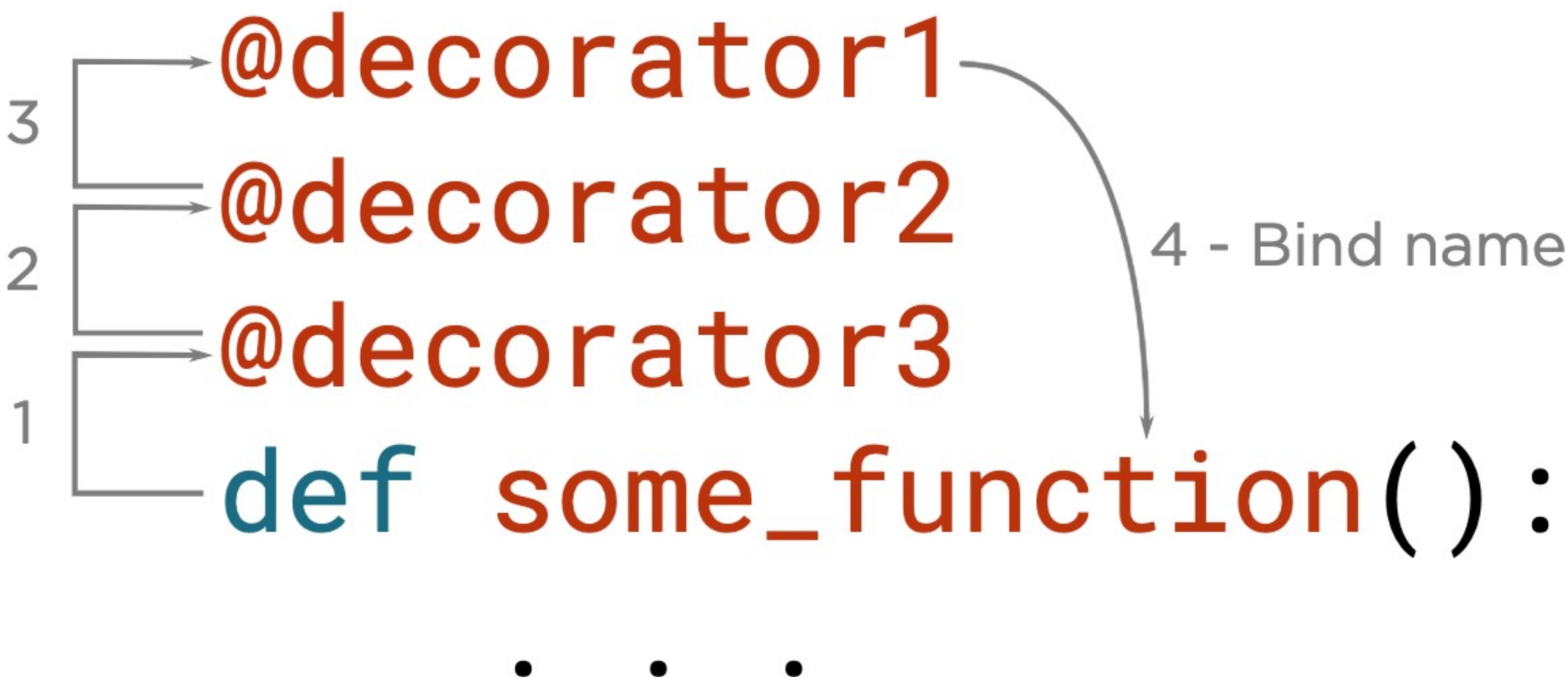
Classes as Decorators

```
>>> class CallCount:
...     def __init__(self, f):
...         self.f = f
...         self.count = 0
...     def __call__(self, *args, **kwargs):
...         self.count += 1
...         return self.f(*args, **kwargs)
...
>>> @CallCount
... def hello(name):
...     print('Hello, {}!'.format(name))
...
>>> hello('Fred')
Hello, Fred!
>>> hello('Wilma')
Hello, Wilma!
>>> hello('Betty')
Hello, Betty!
>>> hello('Barney')
Hello, Barney!
>>> hello.count
4
>>>
```

VTI Academy

Multiple Decorators

Multiple Decorators



VTI Academy

Decorating Methods

Decorating Methods

```
>>> class IslandMaker:
...     def __init__(self, suffix):
...         self.suffix = suffix
...     @tracer
...     def make_island(self, name):
...         return name + self.suffix
...
>>> im = IslandMaker(' Island')
>>> im.make_island('Python')
Calling <function IslandMaker.make_island at 0x10b4bc1f0>
'Python Island'
>>> im.make_island('Llama')
Calling <function IslandMaker.make_island at 0x10b4bc1f0>
'Llama Island'
>>>
```

map()

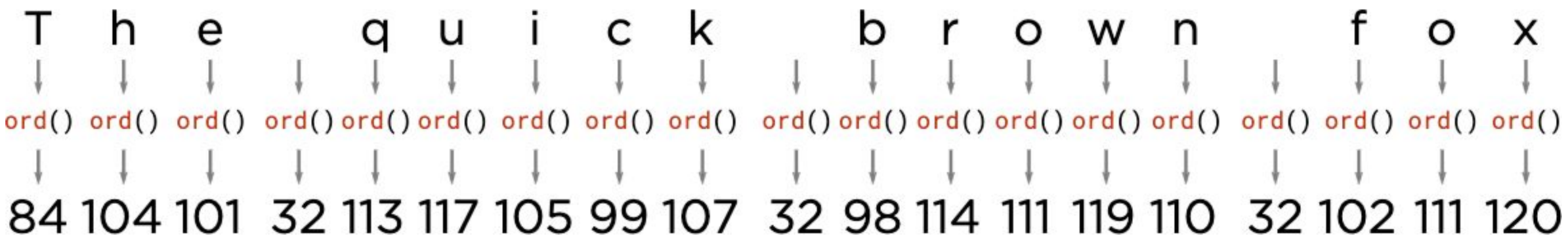
map()

Calls a function for the elements in a sequence, producing a new sequence with the return values

It "maps" a function over a sequence

map()

```
map(ord, 'The quick brown fox')
```



map()

```
>>> map(ord, 'The quick brown fox')
<map object at 0x102ed20d0>
>>>
```

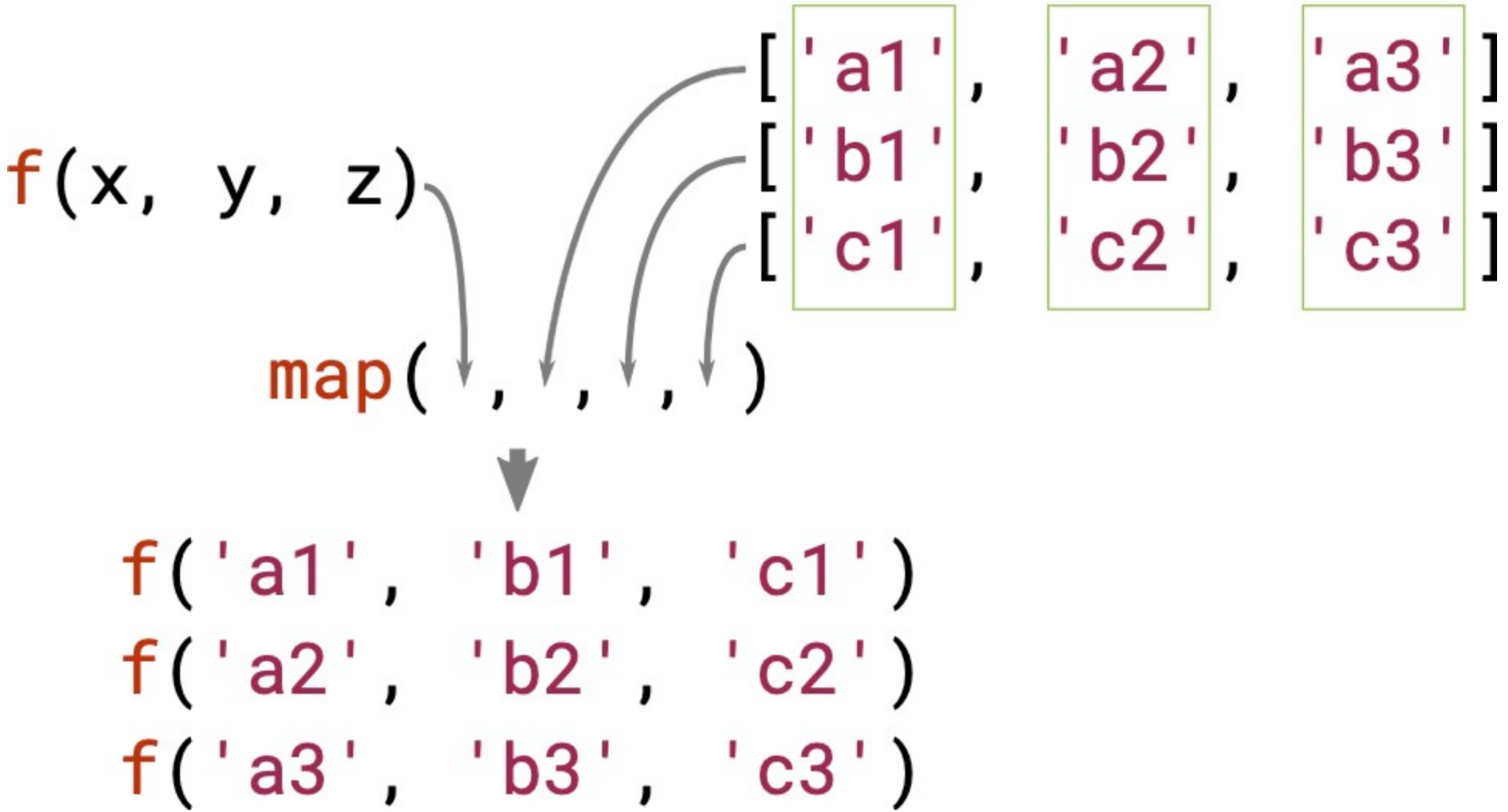
Map() Is Lazy

`map()` will not call its function or access its iterables until they're needed for output

A map object is itself iterable; iterate over it to produce output

map() can be used with as many input sequences as your mapped function needs.

map() with Multiple Iterables



map() with Multiple Iterables

```
>>> sizes = ['small', 'medium', 'large']
>>> colors = ['lavender', 'teal', 'burnt orange']
>>> animals = ['koala', 'platypus', 'salamander']
>>> def combine(size, color, animal):
...     return '{} {} {}'.format(size, color, animal)
...
>>> list(map(combine, sizes, colors, animals))
['small lavender koala', 'medium teal platypus', 'large burnt orange salamander'
]
>>> def combine(quantity, size, color, animal):
...     return '{} x {} {} {}'.format(quantity, size, color, animal)
...
>>> import itertools
>>> list(map(combine, itertools.count(), sizes, colors, animals))
['0 x small lavender koala', '1 x medium teal platypus', '2 x large burnt orange
 salamander']
>>>
```

filter()

filter()

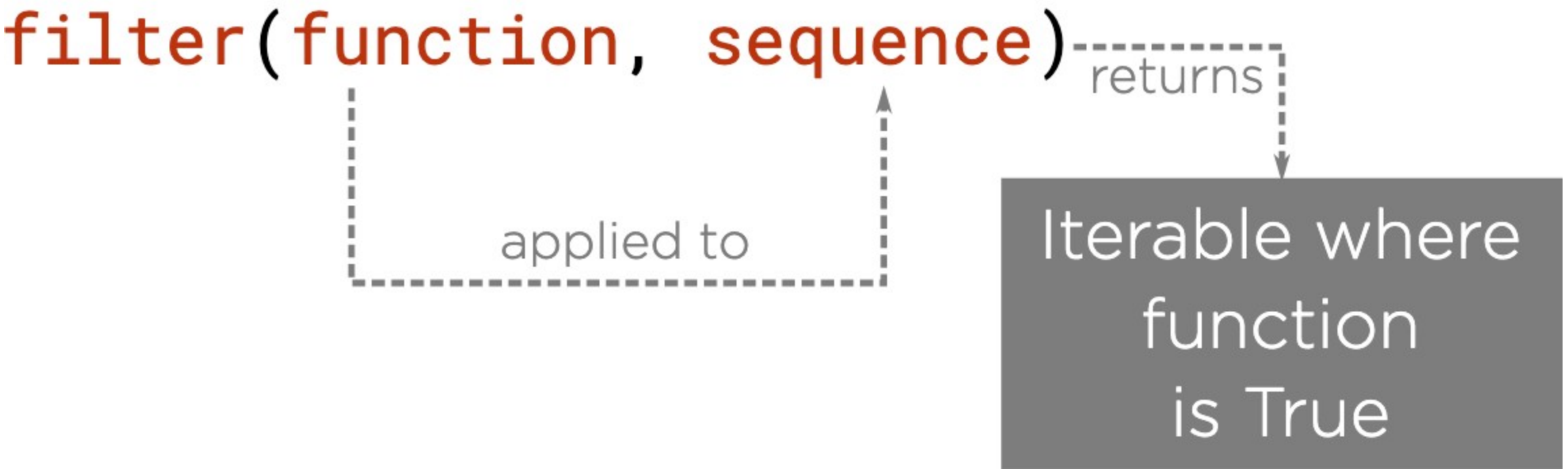
Removes elements from a sequence which don't meet some criteria

Applies a predicate function to each element

Produces its results lazily

Only accepts a single input sequence, and the function must accept only one argument

filter()



filter()

```
>>> positives = filter(lambda x: x > 0, [1, -5, 0, 6, -2, 8])
>>> positives
<filter object at 0x10fe9d490>
>>> list(positives)
[1, 6, 8]
>>>
```

Passing `None` as the first argument to `filter()` will filter out input elements which evaluate to `False`.

Filtering with None

```
>>> trues = filter(None, [0, 1, False, True, [], [1, 2, 3], '', 'hello'])
>>> list(trues)
[1, True, [1, 2, 3], 'hello']
>>>
```

chain()

```
from itertools import chain

l1 = [1, 2, 3]

l2 = [4, 5, 6]

l3 = [7, 8, 9]

for i in chain(l1, l2, l3):

    print(i) # 1,2,3,4,5,6,7,8,9
```

zip()

```
lst1 = [1, 2, 3]
lst2 = [4, 5, 6]
lst3 = [7, 8, 9]
for i in zip(lst1, lst2, lst3):
    print(i)
```

```
Output:
(1, 4, 7)
(2, 5, 8)
(3, 6, 9)
```

reduce()

```
from functools import reduce
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9]
res = reduce(lambda x, y: x + y, numbers)
print(res)
```

```
Output:
45
```

Multi-input Comprehensions

Comprehensions

```
>>> l = [i * 2 for i in range(10)]
>>> d = {i: i * 2 for i in range(10)}
>>> type(d)
<class 'dict'>
>>> s = {i for i in range(10)}
>>> type(s)
<class 'set'>
>>> g = (i for i in range(10))
>>> type(g)
<class 'generator'>
>>>
```

Comprehensions can have multiple input iterables and if-clauses.

Multi-input Comprehensions

```
>>> [(x, y) for x in range(5) for y in range(5)]
[(0, 0), (0, 1), (0, 2), (0, 3), (0, 4), (1, 0), (1, 1), (1, 2), (1, 3), (1, 4),
 (2, 0), (2, 1), (2, 2), (2, 3), (2, 4), (3, 0), (3, 1), (3, 2), (3, 3), (3, 4),
 (4, 0), (4, 1), (4, 2), (4, 3), (4, 4)]
>>> points = []
>>> for x in range(5):
...     for y in range(5):
...         points.append((x, y))
...
>>>
```

Multiple if-clauses

```
>>> values = [x / (x - y)
...           for x in range(100)
...           if x > 50
...           for y in range(100)
...           if x - y != 0]
>>> values = []
>>> for x in range(100):
...     if x > 50:
...         for y in range(100):
...             if x - y != 0:
...                 values.append(x / (x - y))
...
>>> [(x, y) for x in range(10) for y in range(x)]
[(1, 0), (2, 0), (2, 1), (3, 0), (3, 1), (3, 2), (4, 0), (4, 1), (4, 2), (4, 3),
 (5, 0), (5, 1), (5, 2), (5, 3), (5, 4), (6, 0), (6, 1), (6, 2), (6, 3), (6, 4),
 (6, 5), (7, 0), (7, 1), (7, 2), (7, 3), (7, 4), (7, 5), (7, 6), (8, 0), (8, 1),
 (8, 2), (8, 3), (8, 4), (8, 5), (8, 6), (8, 7), (9, 0), (9, 1), (9, 2), (9, 3),
 (9, 4), (9, 5), (9, 6), (9, 7), (9, 8)]
>>> result = []
>>> for x in range(10):
...     for y in range(x):
...         result.append((x, y))
...
>>>
```

Nested Comprehensions

Nested Comprehensions

```
>>> vals = [[y * 3 for y in range(x)] for x in range(10)]
>>> outer = []
>>> for x in range(10):
...     inner = []
...     for y in range(x):
...         inner.append(y * 3)
...     outer.append(inner)
...
>>> vals
[[], [0], [0, 3], [0, 3, 6], [0, 3, 6, 9], [0, 3, 6, 9, 12], [0, 3, 6, 9, 12, 15
], [0, 3, 6, 9, 12, 15, 18], [0, 3, 6, 9, 12, 15, 18, 21], [0, 3, 6, 9, 12, 15,
18, 21, 24]]
>>>
```