Extended Argument and Call Syntax

Austin Bingham
COFOUNDER - SIXTY NORTH

@austin_bingham

Robert Smallshire
COFOUNDER - SIXTY NORTH

@robsmallshire

Overview

Extended argument syntax

Arbitrary numbers of positional arguments

Arbitrary keyword arguments

Positional-only and keyword-only arguments

Extended call syntax

Forwarding arbitrary function arguments

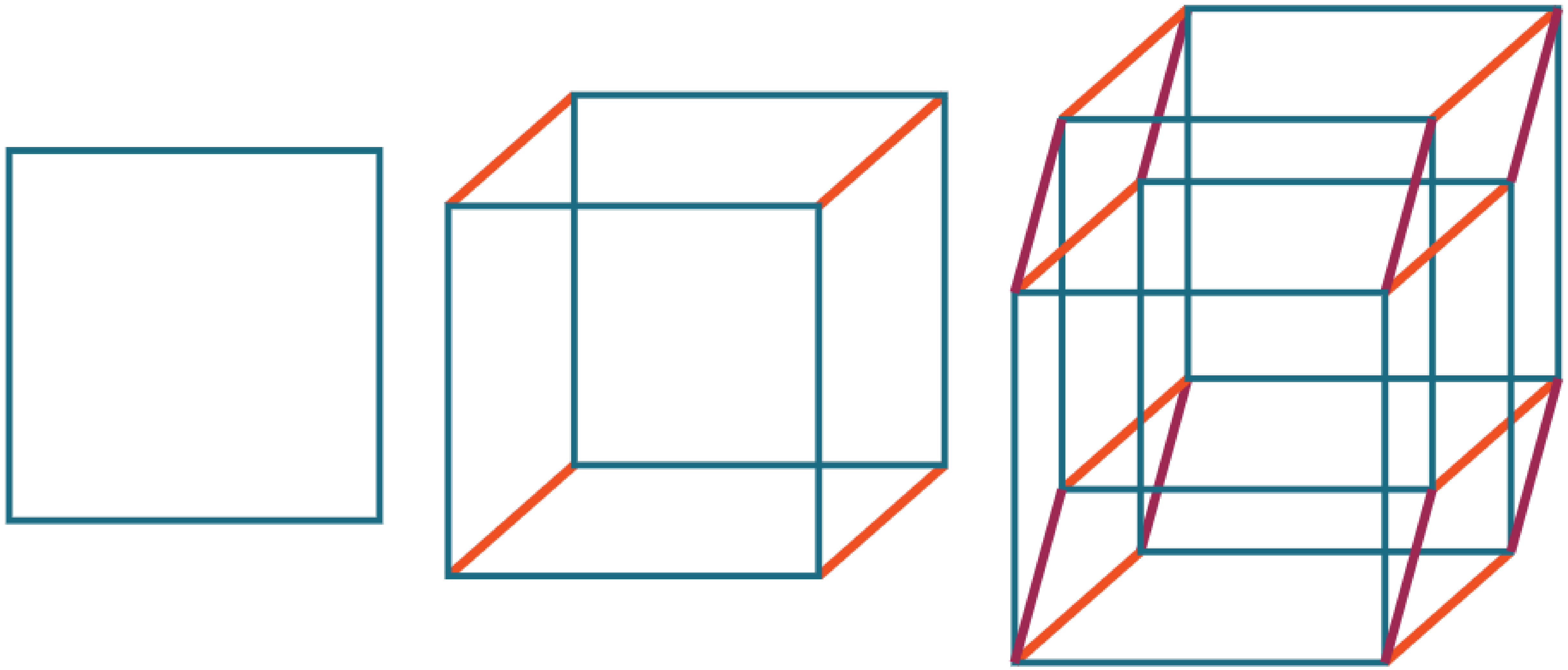
Extended Argument Syntax

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'
>>>
```

Cuboid Volumes



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
>>>
```

Translate the Exception

Catch the `StopIteration` exception thrown by `next()` .

Translate it into the more predictable `TypeError` .

Hypervolume

```
>>> def hypervolume(length, *lengths):
...     v = length
...     for item in lengths:
...         v *= item
...     return v
...
>>> hypervolume(3, 5, 7, 9)
945
>>> hypervolume(3, 5, 7)
105
>>> hypervolume(3, 5)
15
>>> hypervolume(3)
3
>>> hypervolume()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: hypervolume() missing 1 required positional argument: 'length'
>>>
```

Variable Positional Arguments

`functools.reduce()`

Use positional arguments with star-args

Rules for `*args`

1. **Must come after normal positional arguments**
2. **Only collects positional arguments**

Arbitrary keyword arguments

Prefix argument with `**` to accept arbitrary keyword arguments

Conventionally called `**kwargs`

HTML Tag Function

```
def tag(name, **kwargs)
```

HTML Tag Function

```
  File "<stdin>", line 1, in <module>
TypeError: name_tag() takes 2 positional arguments but 3 positional arguments (a
nd 1 keyword-only argument) were given
>>> def print_args(arg1, arg2, *args, kwarg1, kwarg2, **kwargs):
...     print(arg1)
...     print(arg2)
...     print(args)
...     print(kwarg1)
...     print(kwarg2)
...     print(kwargs)
...
>>> print_args(1, 2, 3, 4, 5, kwarg1=6, kwarg2=7, kwarg3=8, kwarg4=9)
1
2
(3, 4, 5)
6
7
{'kwarg3': 8, 'kwarg4': 9}
>>> def print_args(arg1, arg2, *args, kwarg1, kwarg2, **kwargs, kwargs99):
  File "<stdin>", line 1
    def print_args(arg1, arg2, *args, kwarg1, kwarg2, **kwargs, kwargs99):
                                                                ^
SyntaxError: invalid syntax
>>>
```

Positional-only Arguments

Positional-only Arguments

```
>>> def number_length(x, /):
...     return len(str(x))
...
>>> number_length(2112)
4
>>> number_length(x=31557600)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: number_length() got some positional-only arguments passed as keyword
arguments: 'x'
>>>
```

Why Positional-only Arguments?

Parity with modules implemented in other languages

Positional-only Arguments in `range()`

```
>>> range(start=1, stop=100)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: range() takes no keyword arguments
>>>
```

Why Positional-only Arguments?

Parity with modules implemented in other languages

Prevent formal argument names from becoming part of the API

- **This prevents dependencies on the names**
- **Useful when the names have no semantic meaning**

Extended argument syntax applies to all types of callables.

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)
>>>
```

`dict()` uses `**kwargs` in its initializer.

We can use this in our previous example instead of a `dict` literal.

Argument Forwarding

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
>>>
```

Summary

Extended argument syntax for accepting arbitrary positional arguments

As well as arbitrary keyword arguments

Specifying keyword-only arguments

Specifying positional-only arguments

Extended call syntax

Perfect argument forwarding