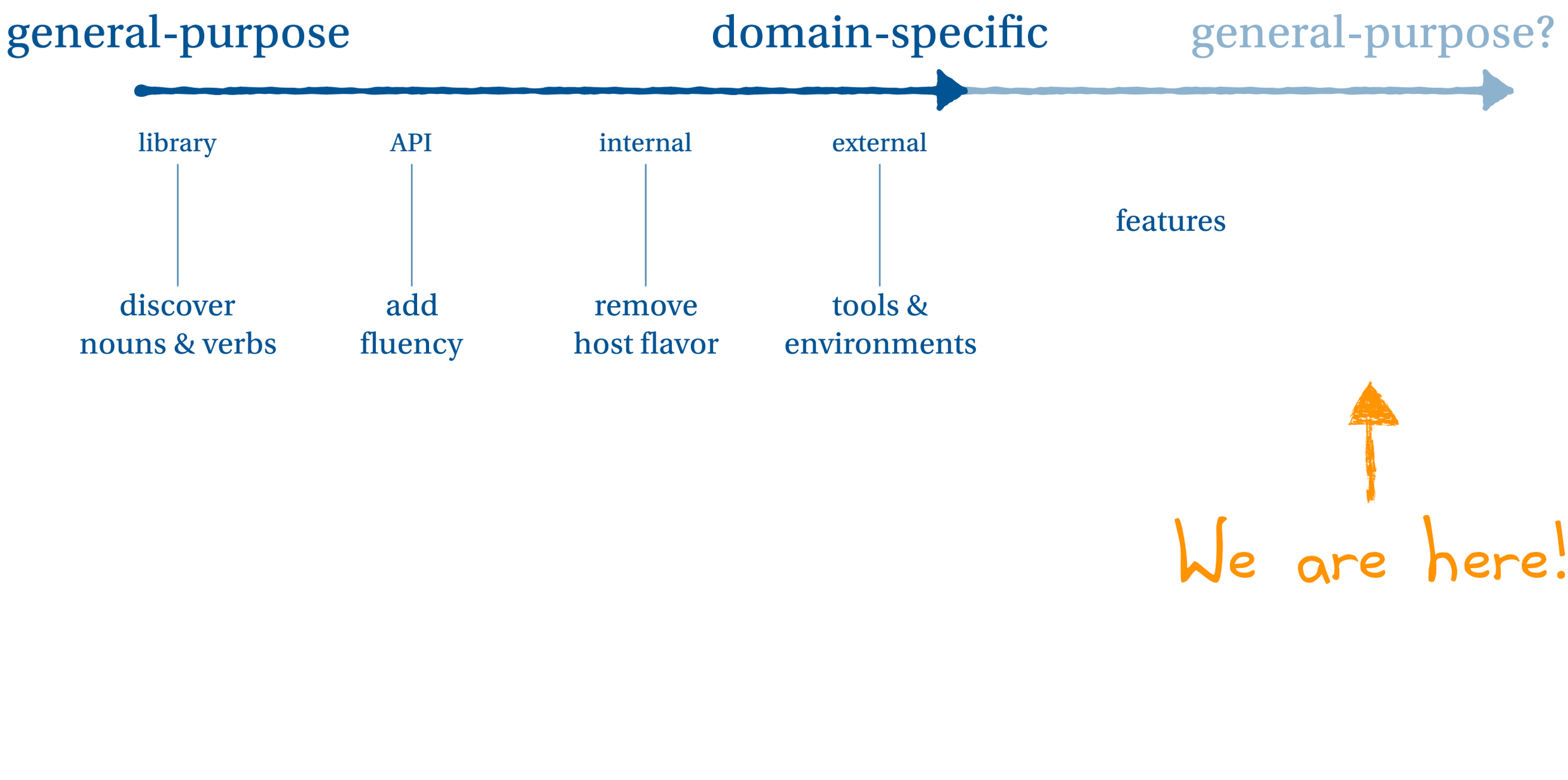
The evolution of a DSL?



A taste of metaprogramming

Python decorators

```
@decorator
def f():
    ...
```

≈

```
def f():
    ...
f = decorator(f)
```

Something more useful

```
def logging(f):

  def wrapper(*args, **kwargs):
    print 'Calling {0} with {1} and {2}'.format(f.func_name, args, kwargs)
    return f(*args, **kwargs)
  return wrapper
```

```
@logging
def f(x,y):
  return x+y
```

A clean integration

```
from functools import wraps

def logging(f):
  @wraps(f)
  def wrapper(*args, **kwargs):
    print 'Calling {0} with {1} and {2}'.format(f.func_name, args, kwargs)
    return f(*args, **kwargs)
  return wrapper



@logging
def f(x,y):
  return x+y
```

Classes as decorators

```
from functools import wraps

class logging(object):
  def __init__(self, handle=sys.stdout):
    self.handle = handle

  def __call__(self, f):
    @wraps(f)
    def wrapper(*args, **kwargs):
      print >> self.handle, \
        'Calling {0} with {1} and {2}'.format(f.func_name, args, kwargs)
      return f(*args, **kwargs)

    return wrapper


@logging(file('log.txt', 'a'))
def g(x,y):
  return x**y
```

Another decorator

```
from functools import wraps

def memoize(f):
  cache = {}

  @wraps(f)
  def wrapper(*args, **kwargs):
    key = (args, tuple(kwargs.values()))
    if key not in cache:
      cache[key] = f(*args, **kwargs)
    return cache[key]

  return wrapper

@memoize
def g(x,y):
  return x**y
```

Decorators are composable

```
@memoize
@logging
def fib(n):
  if n==0 or n==1:
    return 1

  return fib(n-1) + fib(n-2)
```

Atominite

Essence of night

Dia-tonic

```
make(p):

    if potion p is not yet in the pantry:
        foreach potion p_i in p's ingredients:
            make(p_i)
        add p to the pantry
        proclaim "Made p!"

    fetch p from the pantry
```

Illuminectar

```
@ingredients([])
def Atominite(): pass

@ingredients([Atominite])
def Bicarbonite(): pass

@ingredients([Bicarbonite])
def CockroachCocktail(): pass

@ingredients([])
def Diatonic(): pass

@ingredients([Atominite, Diatonic])
def EssenceOfNight(): pass

@ingredients([EssenceOfNight])
def Florabinite(): pass

@ingredients([Diatonic])
def Grapplegum(): pass
```

- Views
- Iterators
- Creating Collections From Scratch
- Conversions Between Java and Scala Collections
- Migrating from Scala 2.7

- The Architecture of Scala Collections
- String Interpolation NEW IN 2.10
- Implicit Classes NEW IN 2.10
- Value Classes and Universal Traits NEW IN 2.10

Reference / Documentation

- Scaladoc
  - Overview
  - Using Scaladoc Effectively
  - Authoring Scaladoc
- Scala REPL
  - Overview

Parallel and Concurrent Programming

- Futures and Promises NEW IN 2.10
- Scala's Parallel Collections Library
  - Overview
  - Concrete Parallel Collection Classes
  - Parallel Collection Conversions
  - Concurrent Tries
  - Architecture of the Parallel Collections Library
  - Creating Custom Parallel Collections
  - Configuring Parallel Collections
  - Measuring Performance
- The Scala Actors Migration Guide NEW IN 2.10
- The Scala Actors API DEPRECATED

Metaprogramming

- Reflection EXPERIMENTAL
  - Overview
  - Environment, Universes, and Mirrors
  - Symbols, Trees, and Types
  - Annotations, Names, Scopes, and More
  - TypeTags and Manifests
  - Thread Safety
  - Changes in Scala 2.11
- Macros EXPERIMENTAL
  - Use Cases
  - Blackbox Vs Whitebox
  - Def Macros
  - Quasiquotes