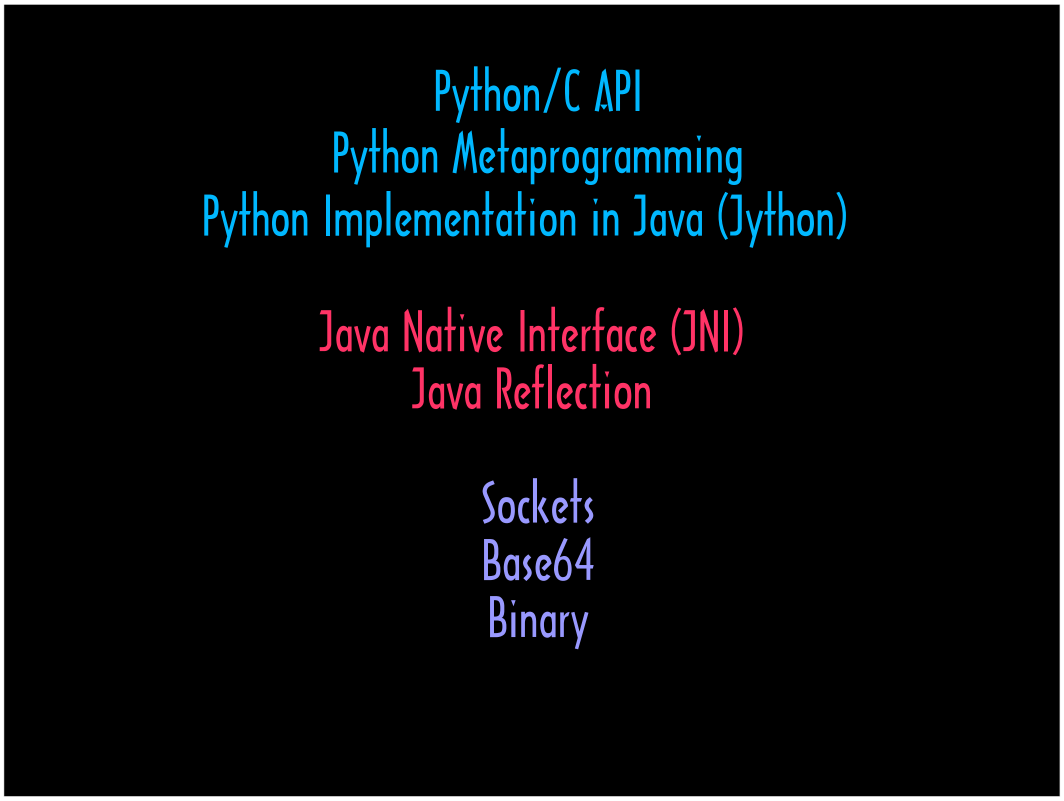
# POLYGLOT PYTHON: PYTHON/SCALA INTEROP

ANDREA O. K. WRIGHT
Chariot Solutions
https://github.com/A-OK/Snakes-and-Ladders
aok@chariotsolutions.com

I don't see this talk as a series of how-to's for developers with a burning need to do Python\Scala interop. It's more of a talk about approaches to integrating a host and guest language, more generally. The API's and libraries I'll be covering all have counterparts in other languages.



Python/C API
Python Metaprogramming
Python Implementation in Java (Jython)

Java Native Interface (JNI)
Java Reflection

Sockets
Base64
Binary

I'll show you basic usage of these APIs, talk a little about what these tools offer beyond what I'll be showing, and we'll also look at how developers have wrapped these APIs in a manner that I like to describe as not just wrapping ...

http://www.wholeliving.com/sites/files/wholeliving.com/ecl/images/content/pub/body_and_soul/2010Q4/mbd106472_1210_gift2_xl.jpg

… but wrapping with a bow.

Scala

Scala is a JVM (Java Virtual Machine) language that compiles down to Java bytecode. There are some incompatibilities, but for the most part, it's easy to access Java from Scala and vice versa.

It probably won't surprise you that many of the tools I'm going to show you were written for Java developers, not Scala developers.

So why did I port all of my examples to Scala?

Because Scala is a multi-paradigm language. It's object-oriented. It has strong support for programming in a functional style, and there are some features unique to Scala that make it powerful, flexible and concise. For all of these reasons, you can use it to ...

http://www.schulzmuseum.org/exhibits/permanent/Christo/christo.html

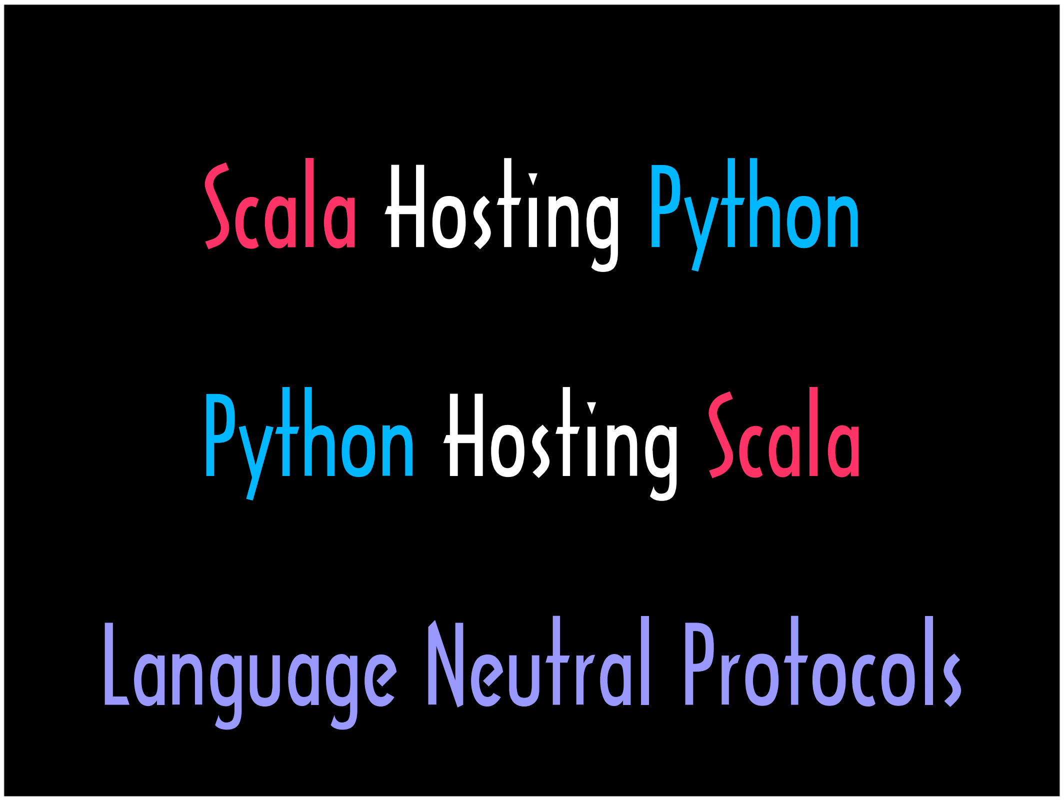
… wrap code in interesting ways.

This is the replica of Snoopy's doghouse, wrapped by the artists Christo and Jeanne-Claude that is currently on display at the Charles M. Schultz Museum In Santa Rosa, California.

http://www.mindfood.com/upload/images/article_images/042009/0e0b6553-5794-410c-80f2-8658288ad749.jpg

And this is a portion of the coast of Australia as wrapped by Christo and Jeanne-Claude. It's part of one of the couple's most epic wrapping projects: wrapping  1 million feet of the Australian coastline.

Between Scala and Python there's great potential for epic wrapping.



Scala Hosting Python

Python Hosting Scala

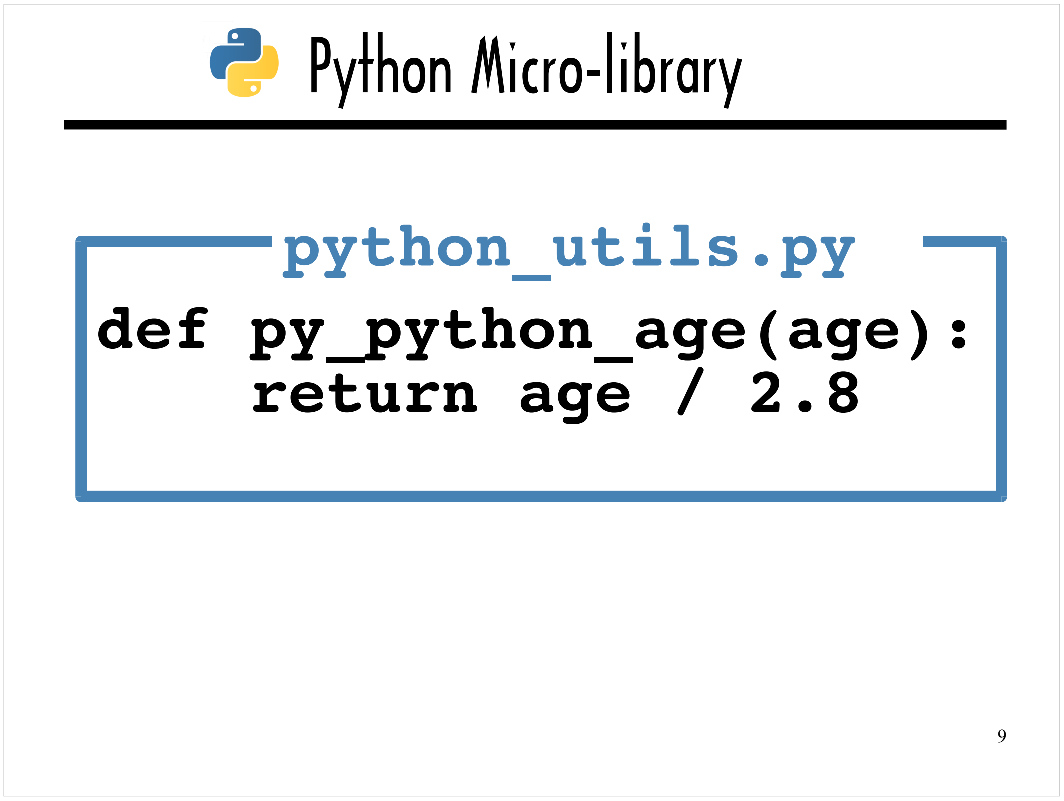
Language Neutral Protocols

I'm going to start off with examples of Scala hosting Python, move on to examples where Python is the host language, and then cover language neutral protocols.

The code examples in the slide generally omit “import” statements and “include” statements to keep the slides from being too cluttered, but the complete source code for the examples can be found at: https://github.com/A-OK/Snakes-and-Ladders

## Scala hosting Python

In this section, we'll look at Scala programs that access functions defined in a Python module.



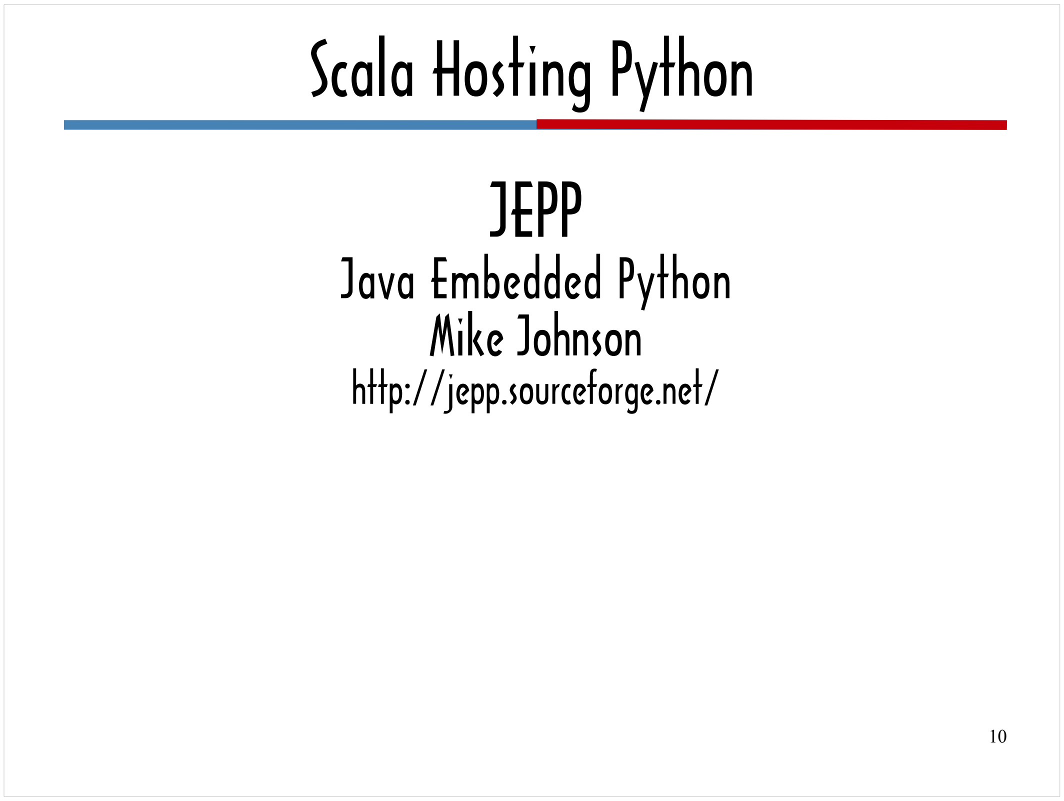
## Python Micro-library

python_utils.py

```
def py_python_age(age):
    return age / 2.8
```

Here's the Python module I will be using in almost all of the examples in this Section. I want to focus on the interop libraries, not some business domain, so I'm using this python_utils micro-library to represent an actual, useful Python -based resource.

The single symbolic function in this micro-library takes a person's age and determines how old that person would be if that person was a python regius (a royal or ball python) . The formula is based on the average life expectancy for a python being 30 years. Using this formula, a 30-year old person would have The relative maturity of a 10.1-year-old python.



## Scala Hosting Python

JEPP

Java Embedded Python

Mike Johnson

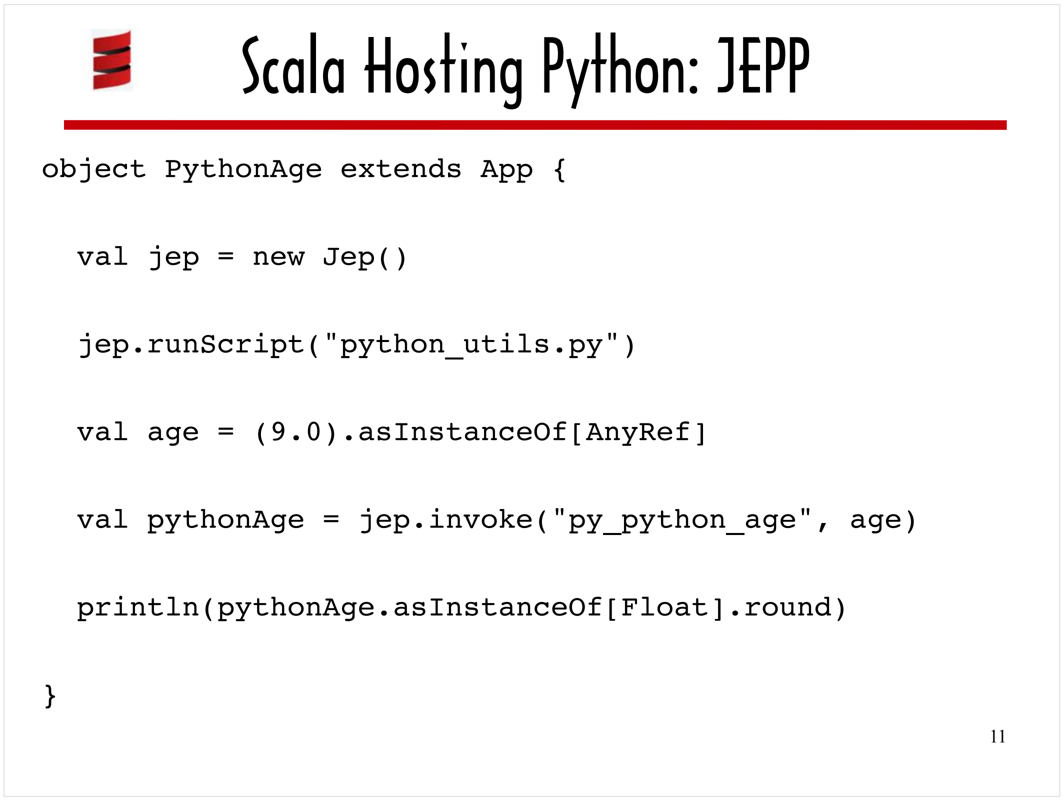
http://jepp.sourceforge.net/

10

The first Python\Java bridge I'm going to use in a Scala program is JEPP. You'l see 'JEPP' with 2 P's and “JEP” with one P used interchangeably in the JEPP documentation and any Web resources about JEP. I learned from its author, Mike Johnson, that this is because there was already a “JEP” project on SourceForge when he initially shared the code from there, and project names across SourceForge need to be unique. The code is currently available on Github, which does not enforce this restriction across accounts.

The JEPP project leverages both the Python C/API, which ships with Python and the Java Native Interface (JNI) API, which ships with Java. JEPP uses the Python C/API to execute logic written in Python from functions written in C, and then it uses Java's JNI to invoke said functions written in C that use the Python/C API to execute logic written in Python.

In the next few slides we'll look at a Scala program that use JEPP to communicate with a Python module, and the slides following that take you behind the scenes to look at some of the relevant C code.



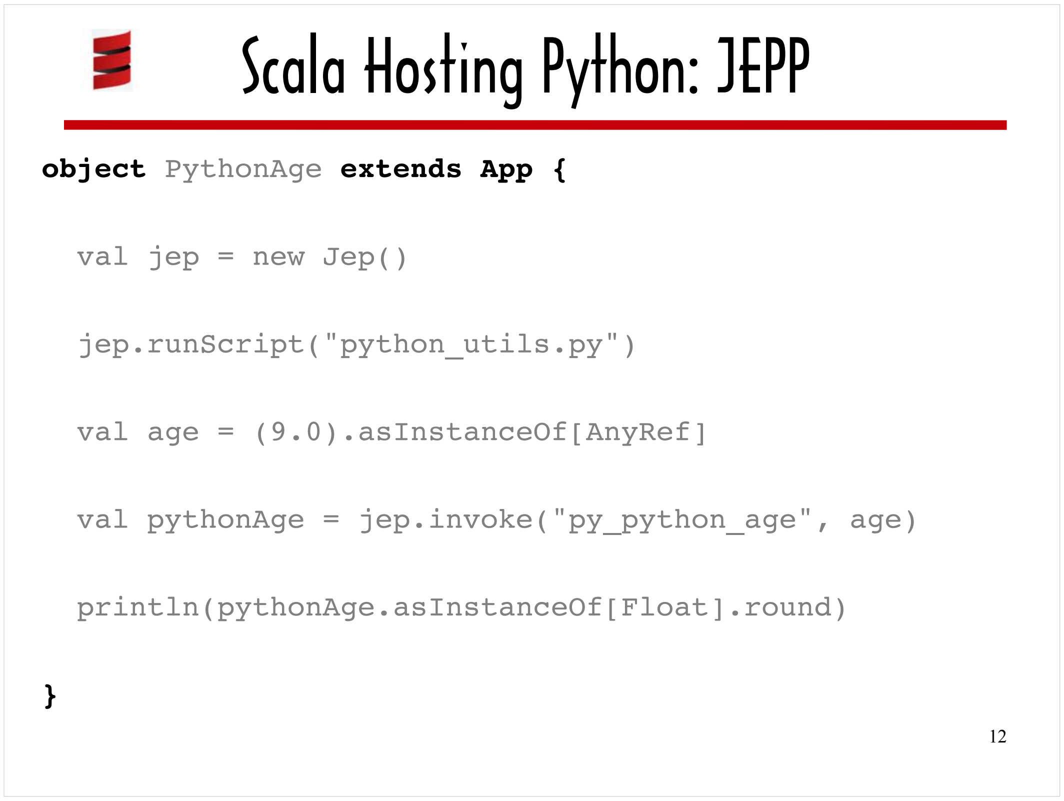
## Scala Hosting Python: JEPP

```
object PythonAge extends App {

  val jep = new Jep()

  jep.runScript("python_utils.py")

  val age = (9.0).asInstanceOf[AnyRef]

  val pythonAge = jep.invoke("py_python_age", age)

  println(pythonAge.asInstanceOf[Float].round)

}
```

11

This slide provides a first look at a Scala program.

This talk does not require previous experience with Scala. Anyone familiar with using dot notation to invoke methods on object should be able to understand most of the Scala on these slides. If there's anything else you need to know about Scala syntax or features, I'll try to to fill you in on an as-needed basis.



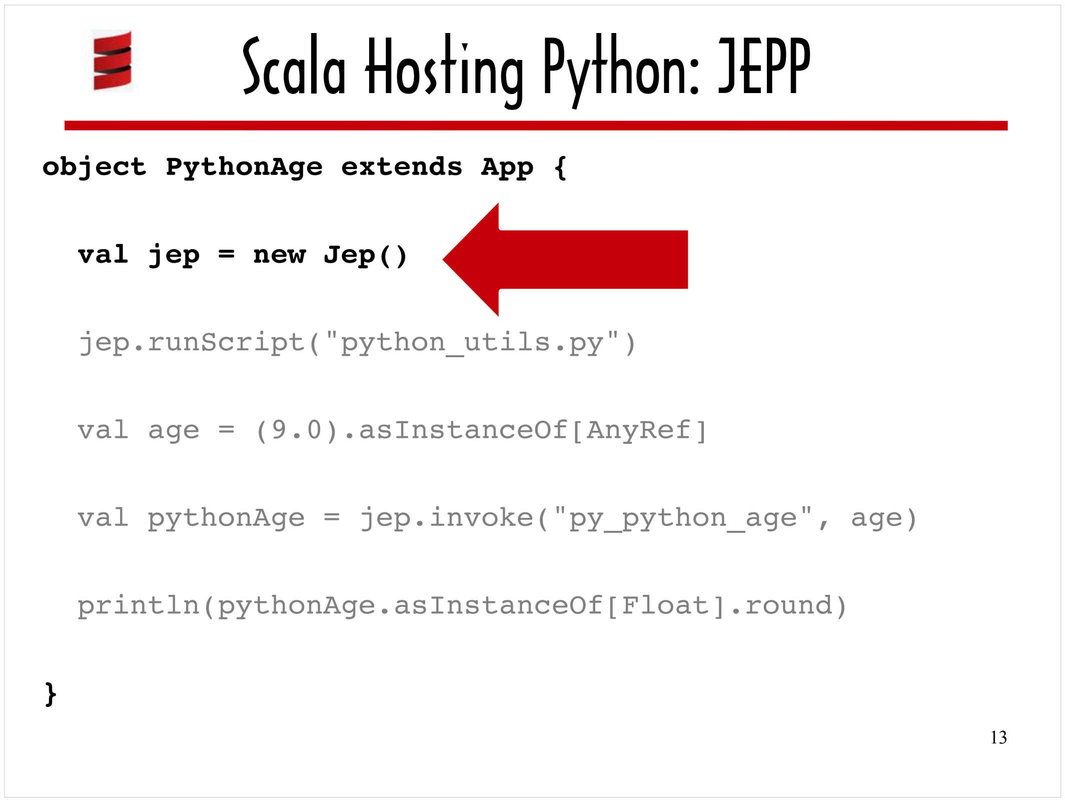
## Scala Hosting Python: JEPP

```
object PythonAge extends App {

  val jep = new Jep()

  jep.runScript("python_utils.py")

  val age = (9.0).asInstanceOf[AnyRef]

  val pythonAge = jep.invoke("py_python_age", age)

  println(pythonAge.asInstanceOf[Float].round)

}
```

This is the definition of a Scala class called **`PythonAge`**.

The keyword **`object`** appears where you might expect to see the **`class`** keyword. The keyword object is used in lieu of class to create a Scala Singleton Object. You can access the properties of and invoke methods on a Singleton Object without instantiating it. I'll talk more about using Singleton Objects when we look at accessing Scala from within Python programs.

Using the phrase **`extends App`** is conceptually related to defining a **`main`** method in a Python module. The Scala compiler will insert a **`main`** method into this class. This **`main`** method will contain all the statements between the curly braces, and will be executed when the file containing the class definition for this class is run from the command line by passing its name to the Scala Interpreter. In other words, this phrase enables you to run this class file as if it were a script.



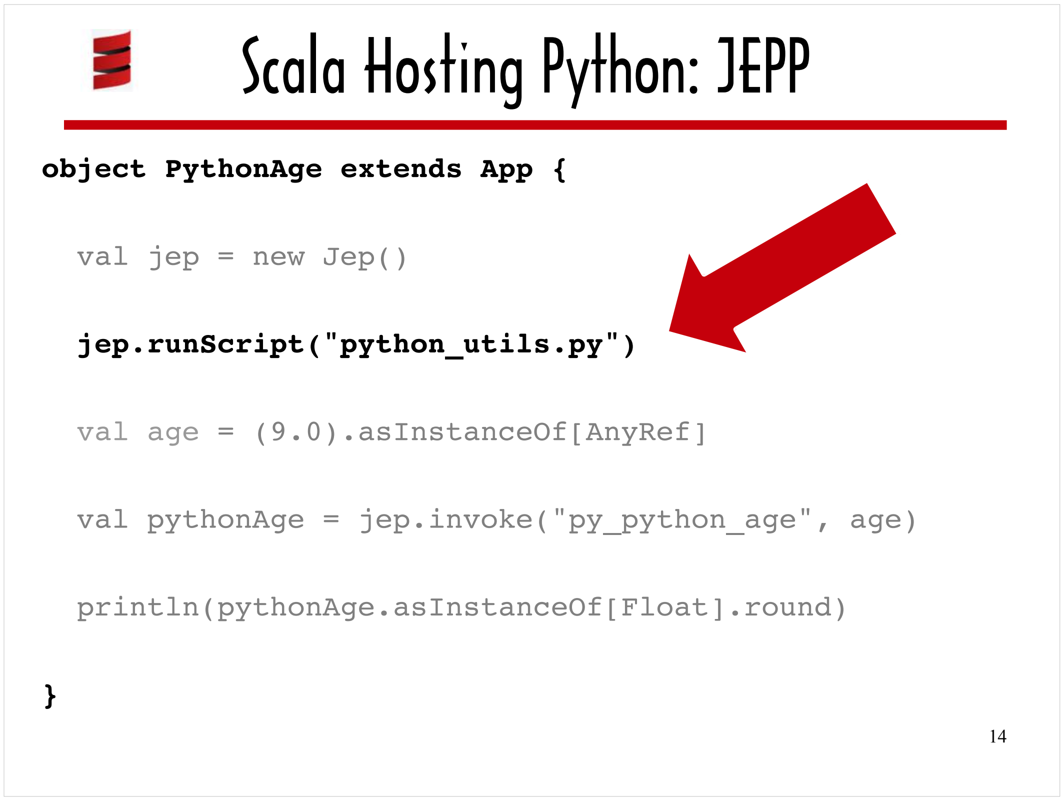
## Scala Hosting Python: JEPP

```
object PythonAge extends App {

  val jep = new Jep()

  jep.runScript("python_utils.py")

  val age = (9.0).asInstanceOf[AnyRef]

  val pythonAge = jep.invoke("py_python_age", age)

  println(pythonAge.asInstanceOf[Float].round)

}
```

This line instantiates a **`Jep`** object. Jep is a Java-based component of the JEPP library. The JEPP library contains Java components, C components and Python components.

All of the services that JEPP provides to Java classes (and that are also accessible to Scala classes) are defined as methods on the **`Jep`** class.

The Scala keyword **`val`**, which begins the line, denotes an immutable variable the Scala keyword **`var`** would be used declare a mutable variable. The Scala community recommends using **`val`**s as opposed to **`var`**s whenever possible.

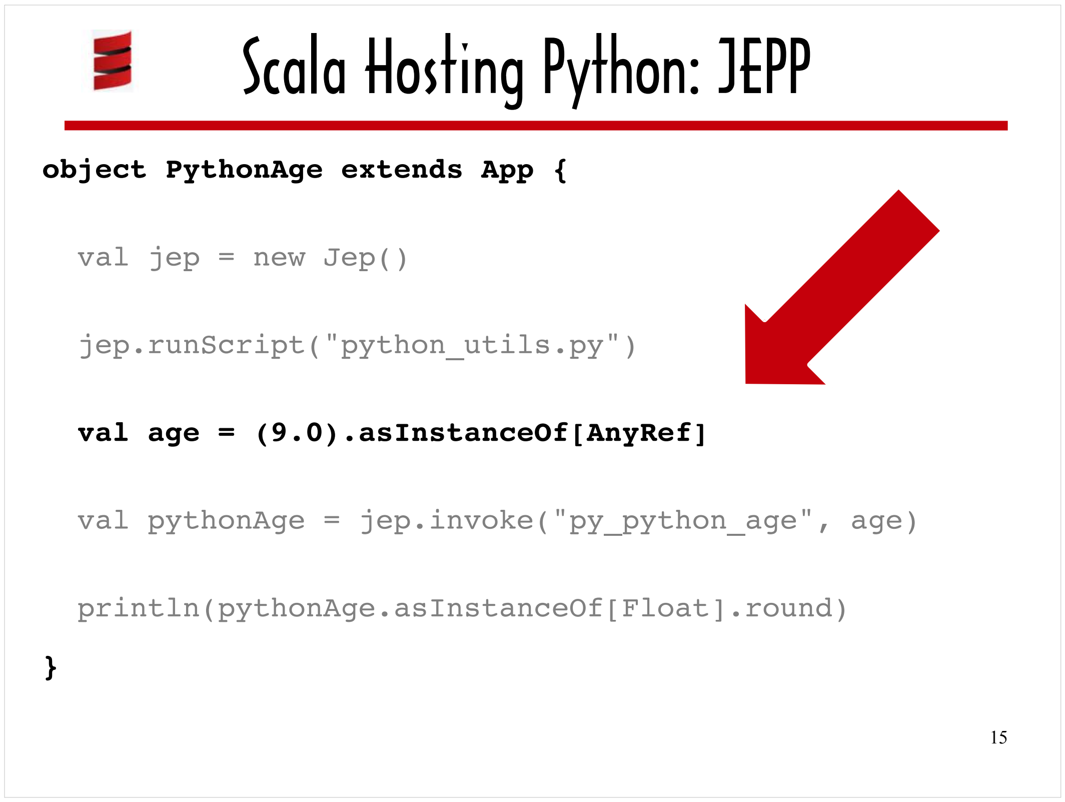


## Scala Hosting Python: JEPP

```
object PythonAge extends App {

  val jep = new Jep()

  jep.runScript("python_utils.py")

  val age = (9.0).asInstanceOf[AnyRef]

  val pythonAge = jep.invoke("py_python_age", age)

  println(pythonAge.asInstanceOf[Float].round)

}
```

14

Here, invoking JEPP's **`runScript`** method and passing it the name of the Python micro-library script loads the script into the JEPP environment, Enabling Scala to access any Python functions, classes or variables defined therein.



## Scala Hosting Python: JEPP

```
object PythonAge extends App {

  val jep = new Jep()

  jep.runScript("python_utils.py")

  val age = (9.0).asInstanceOf[AnyRef]

  val pythonAge = jep.invoke("py_python_age", age)

  println(pythonAge.asInstanceOf[Float].round)

}
```

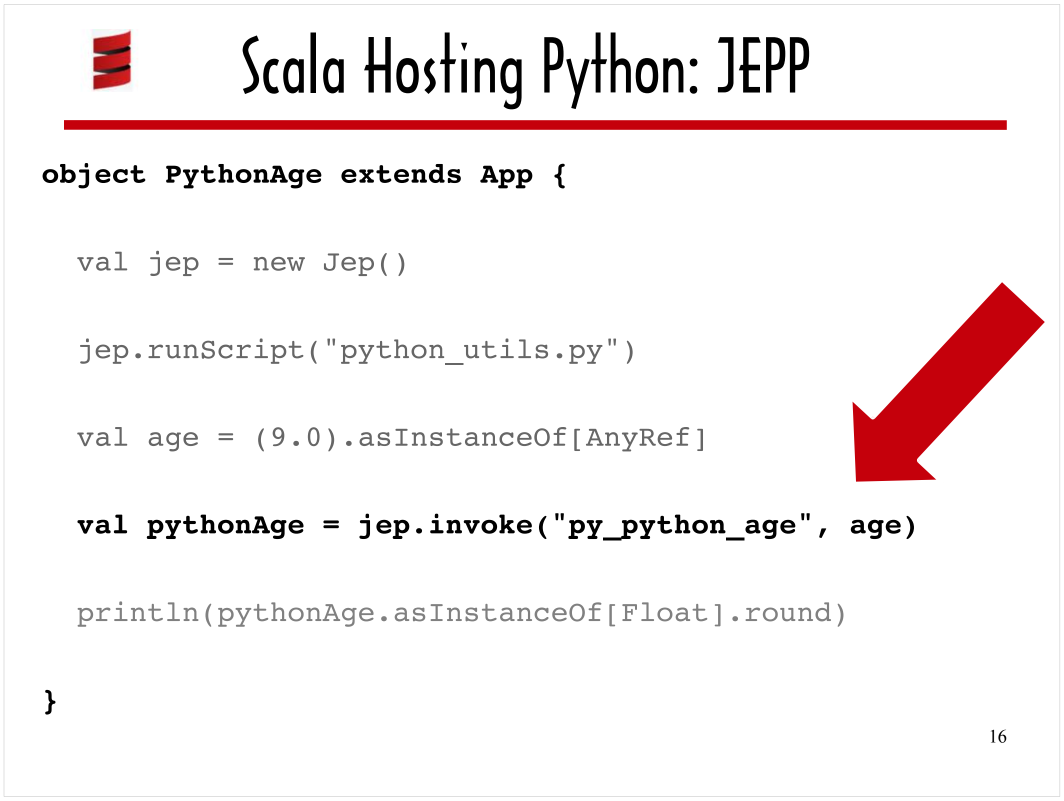
15

Here we're preparing an argument to pass to the JEPP **`invoke`** method and assigning it to a variable called **`age`**.

The JEPP **`invoke`** method takes the name of the Python function you want to invoke and a collection of arguments to pass to that Python function.

The arguments must each be cast as Java objects (java.lang.Object or its subclasses). The Scala phrase **`asInstanceOf[AnyRef]`** takes care of this for the single argument I am going to pass – the age I want to convert to python years. Scala's “AnyRef” is comparable to “java.lang.Object”.

I'm passing the number 9.0 because that is how old the Scala language is at this writing.

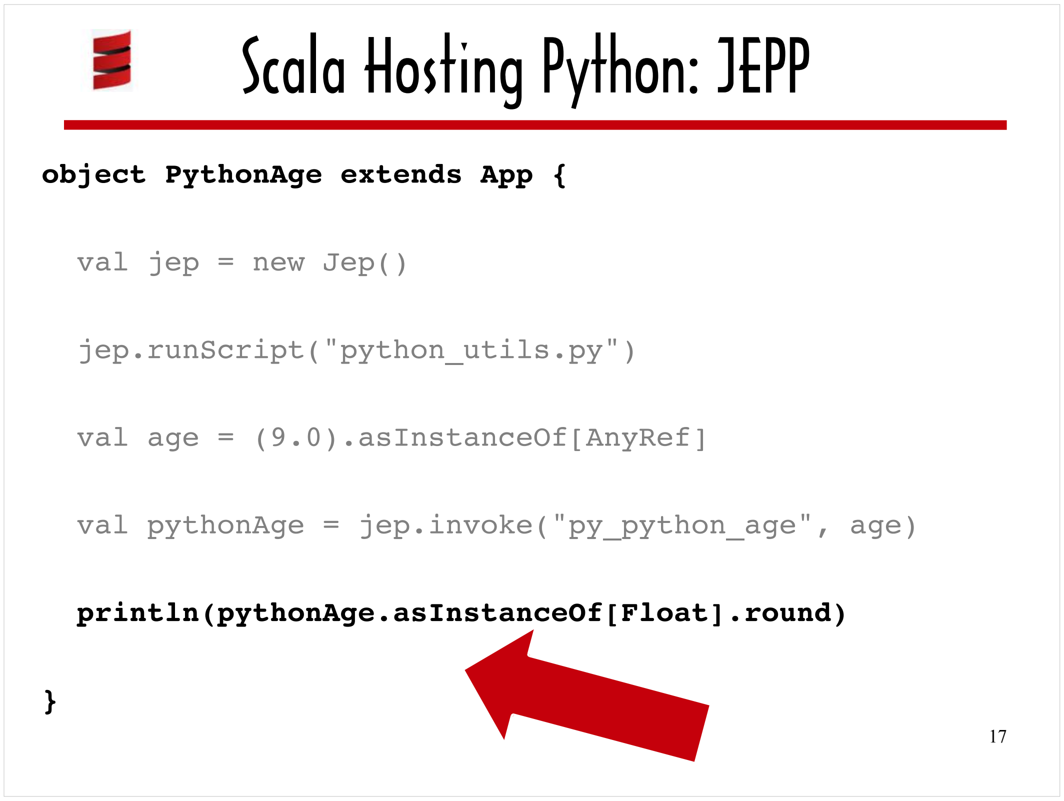


## Scala Hosting Python: JEPP

```
object PythonAge extends App {

  val jep = new Jep()

  jep.runScript("python_utils.py")

  val age = (9.0).asInstanceOf[AnyRef]

  val pythonAge = jep.invoke("py_python_age", age)

  println(pythonAge.asInstanceOf[Float].round)

}
```

16

Here I'm using the JEPP **`invoke`** method to invoke the Python micro-library function **`py_python_age`** and assigning the result to the variable **`pythonAge.`**



## Scala Hosting Python: JEPP

```
object PythonAge extends App {

  val jep = new Jep()

  jep.runScript("python_utils.py")

  val age = (9.0).asInstanceOf[AnyRef]

  val pythonAge = jep.invoke("py_python_age", age)

  println(pythonAge.asInstanceOf[Float].round)

}
```

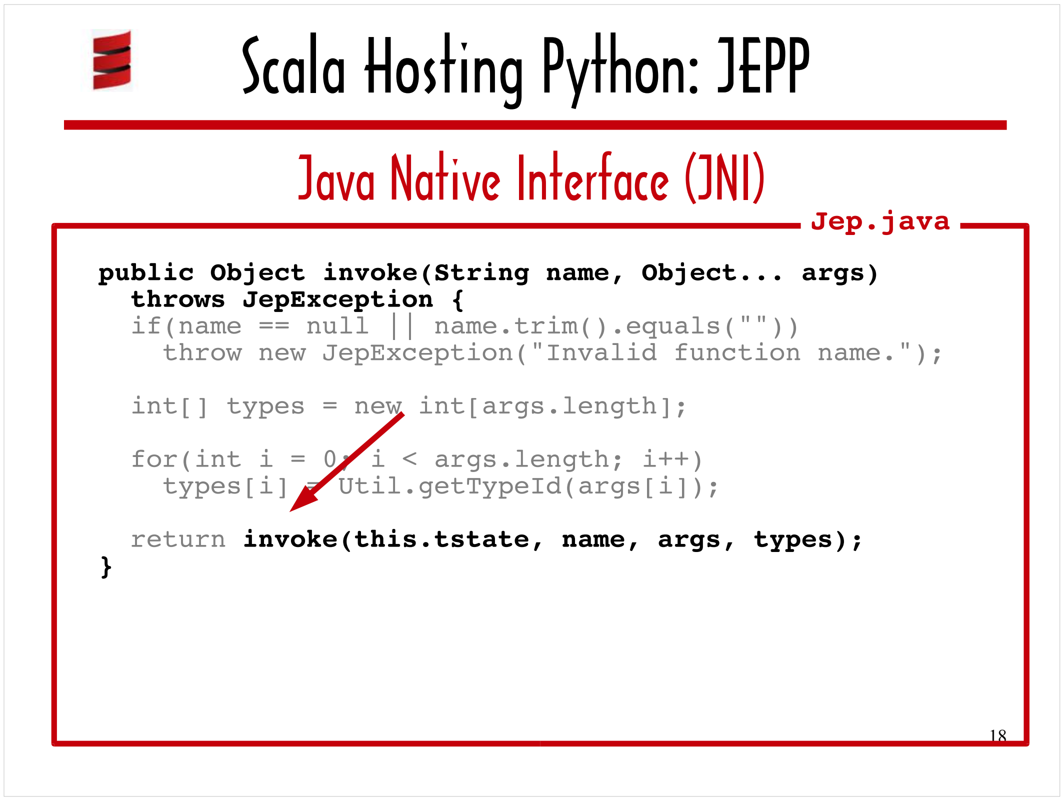
17

This line prints out the age in python years (the value of **`pythonAge`**) after rounding it.

The reason I'm not just printing out **`pythonAge`** is because any generic object can be passed to **`println`** in Scala. I've seen a lot of language interop Sample code online that simply prints out a result once it's obtained from a library written in the guest language.

Calling **`round`** represents doing something useful with the return value, beyond what you can do with a generic object in Scala.

In this  case, I need to explicitly use Scala's **`asInstanceOf`** to cast the return value as a Float in order to round it. In some examples in this talk, the bridge library calls will take care of this casting. I want to give you an idea of where you can expect to do casting in your own interop code.



## Scala Hosting Python: JEPP

### Java Native Interface (JNI)

Jep.java

```
public Object invoke(String name, Object... args)
  throws JepException {
  if(name == null || name.trim().equals(""))
    throw new JepException("Invalid function name.");

  int[] types = new int[args.length];

  for(int i = 0; i < args.length; i++)
    types[i] = Util.getTypeId(args[i]);

  return invoke(this.tstate, name, args, types);
}
```

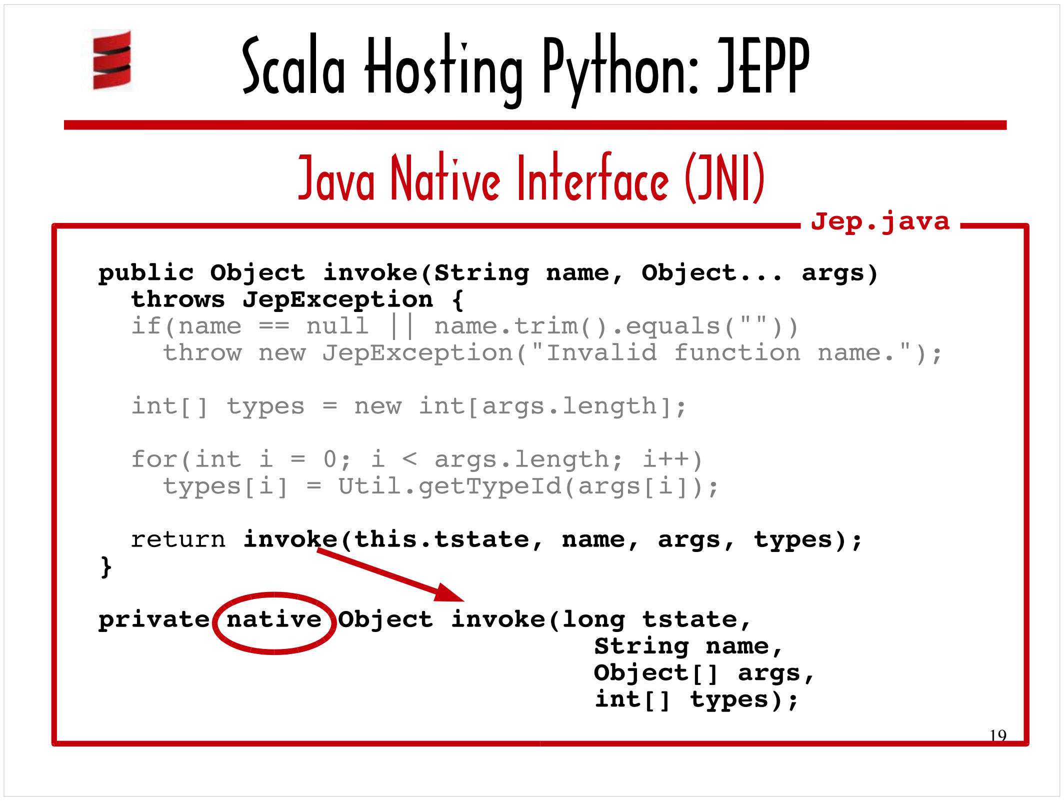
18

Now that we have looked at how to use JEPP to invoke a Python function from a Scala program, we'll look at how JEPP wraps the Python C/API and JNI in order to accomplish this.

Here is the source code for JEPP's **`invoke`**, which is defined as a method in the **`Jep`** class.

I've bolded and I'm using an arrow to point out a call to a different **`invoke`** method with a different signature. In Java (and Scala) methods you can define the same method multiple times using the same method name with different arguments.

This second **`invoke`** method ...



## Scala Hosting Python: JEPP

### Java Native Interface (JNI)

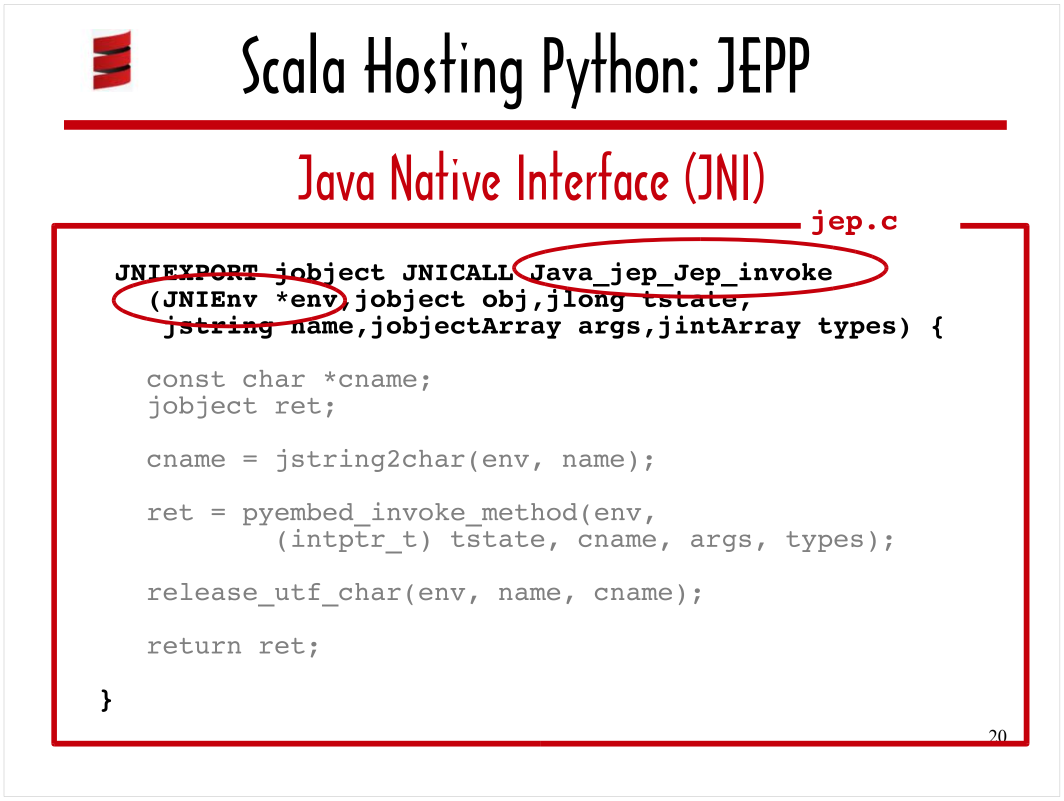
Jep.java

```
public Object invoke(String name, Object... args)
  throws JepException {
  if(name == null || name.trim().equals(""))
    throw new JepException("Invalid function name.");

  int[] types = new int[args.length];

  for(int i = 0; i < args.length; i++)
    types[i] = Util.getTypeId(args[i]);

  return invoke(this.tstate, name, args, types);
}

private native Object invoke(long tstate,
                              String name,
                              Object[] args,
                              int[] types);
```

19

… is declared as a native method, meaning it is implemented in C and accessible via JNI.

The bottom part of this slide shows how this nested invoke (with an arity of 4) is declared as a native method in **`Jep.java`** using the **`native`** keyword.



## Scala Hosting Python: JEPP

### Java Native Interface (JNI)

jep.c

```
JNIEXPORT jobject JNICALL Java_jep_Jep_invoke
  (JNIEnv *env,jobject obj,jlong tstate,
   jstring name,jobjectArray args,jintArray types) {

    const char *cname;
    jobject ret;

    cname = jstring2char(env, name);

    ret = pyembed_invoke_method(env,
            (intptr_t) tstate, cname, args, types);

    release_utf_char(env, name, cname);

    return ret;

}
```

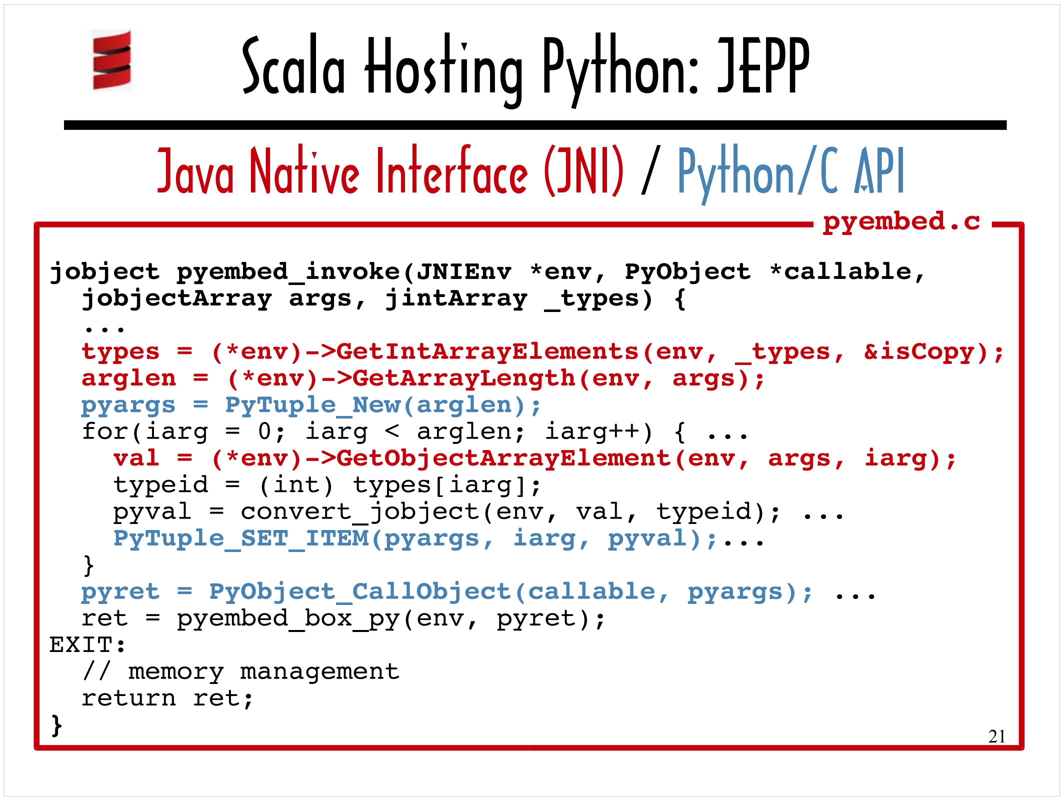
20

This is the C implementation that corresponds to the Java native declaration. It lives in the file **`jep.c`**.

The name of the function, which I have circled, follows the JNI naming conventions for C functions – the word 'Java' followed by the fully qualified name of the Java class (ie including the package name), followed by the function name : **`Java_jep_Jep_invoke`**

I've also circled the first argument because I want to point out that a JNIEnv interface pointer, which provides access to all the functions JNI offers, is passed in to every native method made available via JNI. It represent a decision that's an important part of the JNI architecture. Because it is passed in as a function argument, as opposed to being configured as a hard-wired reference, developers can run JNI-based programs against a different JVM implementation than they built their libraries against.

In the next slide, we'll look at one of the functions that is called by the logic in **`Java_jep_Jep_invoke`** downstream. I'm showing you this particular function...



## Scala Hosting Python: JEPP

Java Native Interface (JNI) / Python/C API

pyembed.c

```
jobject pyembed_invoke(JNIEnv *env, PyObject *callable,
  jobjectArray args, jintArray _types) {
  ...
  types = (*env)->GetIntArrayElements(env, _types, &isCopy);
  arglen = (*env)->GetArrayLength(env, args);
  pyargs = PyTuple_New(arglen);
  for(iarg = 0; iarg < arglen; iarg++) { ...
    val = (*env)->GetObjectArrayElement(env, args, iarg);
    typeid = (int) types[iarg];
    pyval = convert_jobject(env, val, typeid); ...
    PyTuple_SET_ITEM(pyargs, iarg, pyval);...
  }
  pyret = PyObject_CallObject(callable, pyargs); ...
  ret = pyembed_box_py(env, pyret);
EXIT:
  // memory management
  return ret;
}
```

21

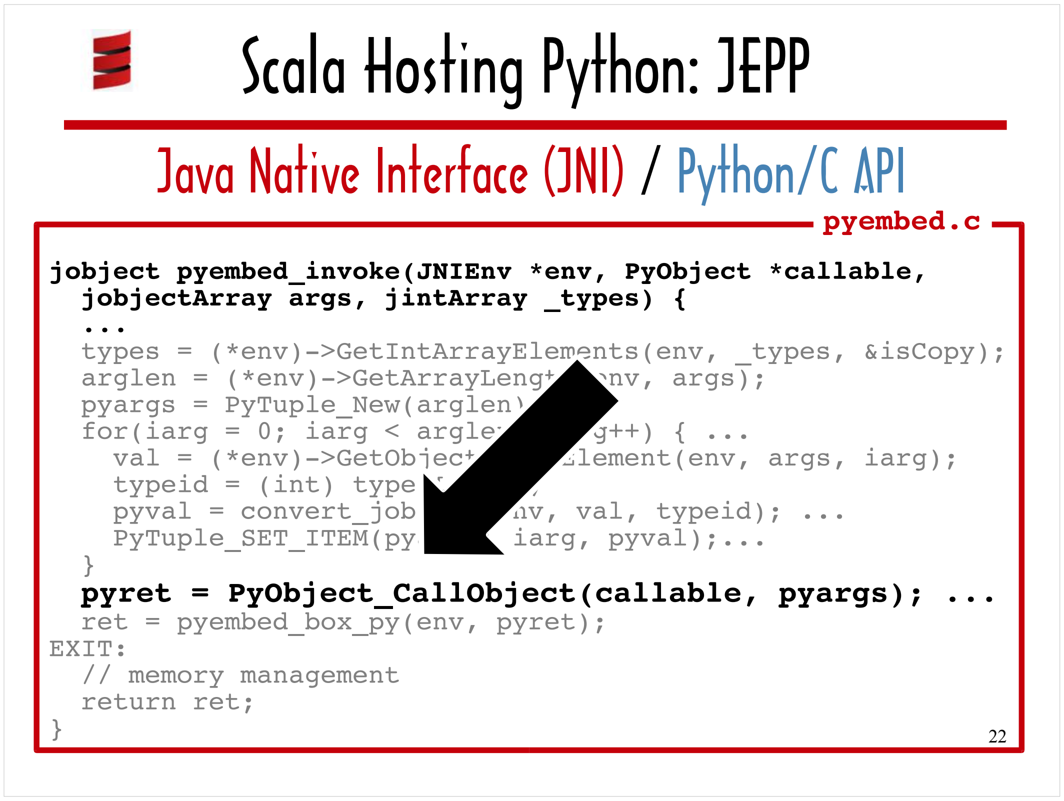
...because it shows JNI and the Python/C API actually talking to each other.

Lines that call into JNI functions are red and lines that call into the Python/C API are blue. Ellipses and comments serve as placeholders for code that handles memory management or exceptions.

Since we're now several layers removed from the top level JEPP API call where we started this deep dive, recall that we're walking through functions called by **`Jep#invoke`**, which invokes a Python function, defined in a **`.py`** file, given the name of the function and a variable-length list of arguments, typecast as Java objects.

JNI calls are used to gain access to the elements in the array containing the arguments that need to be passed to the Python function we want to invoke via JEPP (ie **`py_python_age`**). Python/C API functions use values that were extracted via JNI to populate the **`pyargs`** tuple, which will get passed to **`PyObject_CallObjec`**t, the Python/C API function that can invoke a Python function given the function name and the function arguments packaged in a tuple.

I consider this call to **`PyObject_CallObjec`**t, which I have ...



## Scala Hosting Python: JEPP

Java Native Interface (JNI) / Python/C API

pyembed.c

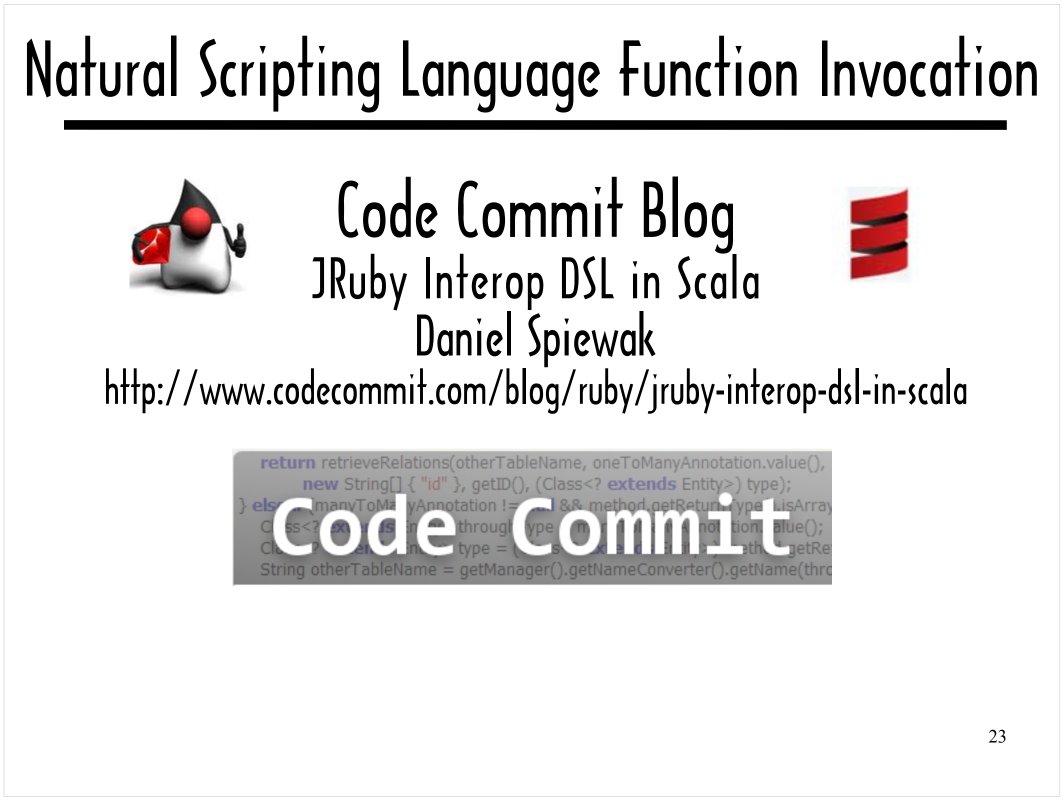
```
jobject pyembed_invoke(JNIEnv *env, PyObject *callable,
  jobjectArray args, jintArray _types) {
  ...
  types = (*env)->GetIntArrayElements(env, _types, &isCopy);
  arglen = (*env)->GetArrayLengt    nv, args);
  pyargs = PyTuple_New(arglen)
  for(iarg = 0; iarg < argle    g++) { ...
    val = (*env)->GetObject     Element(env, args, iarg);
    typeid = (int) type
    pyval = convert_job       nv, val, typeid); ...
    PyTuple_SET_ITEM(py       iarg, pyval);...
  }
  pyret = PyObject_CallObject(callable, pyargs); ...
  ret = pyembed_box_py(env, pyret);
EXIT:
  // memory management
  return ret;
}
```

… highlighted in this view of the function we were just looking at, to be the focal point of JEPP. The fact Python is flexible enough to offer a Python/C API call that invokes arbitrary functions defined in a **`.py`** file gets right to the heart of JEPP's *raison d'etre.*

If I wanted to call functions from a C utilities library from a Scala program using JNI alone, I'd need to either write an individual JNI-based wrapper for each and every function in that C utilities library or I'd have to write some kind of home-grown dispatcher that would need to know at least some details about every C function I want to expose to Scala.

Every time a new function was added to said C utilities library I would need to write a new JNI-based wrapper or I'd have to modify my dispatcher code – and I'd have to do some recompilation.

With JEPP, on the other hand, if I want to call functions from a Python utilities library from Scala, I can call any of the Python functions via **`Jep#invoke`**. When new functions are added to the Python utilities library, they are immediately available for use from Scala right after I save my **`.py`** file.



## Natural Scripting Language Function Invocation

Code Commit Blog
JRuby Interop DSL in Scala
Daniel Spiewak
http://www.codecommit.com/blog/ruby/jruby-interop-dsl-in-scala

Code Commit

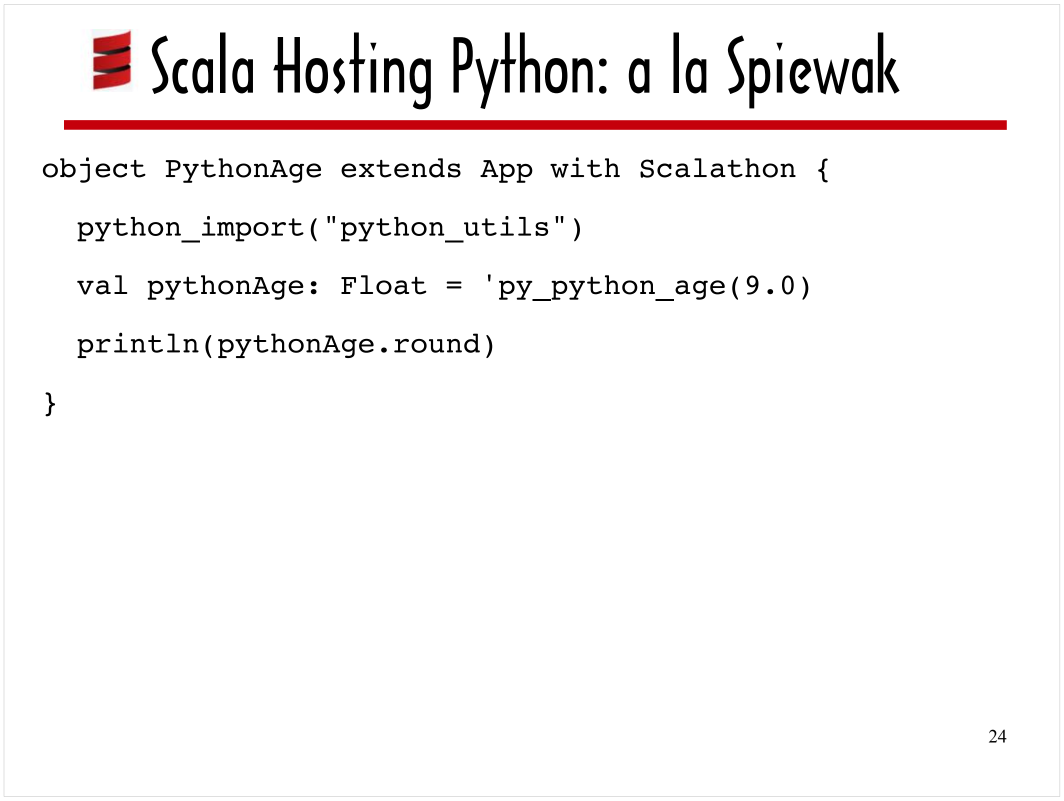
23

While using JEPP enables you to write pure, unfettered Python for consumption by Scala, the code you need to write on the Scala side, including packaging arguments to pass to the Python function in an array and using **`Jep#invoke`**, is a bit cumbersome.

Daniel Spiewak, a programming languages enthusiast and blogger, came up with an idea for how to wrap JEPP-like library calls so that calling a function written in a guest language in a Scala program would be virtually indistinguishable from calling a Scala function.

He introduced this idea on his blog, Code Commit, using JRuby as the guest language in all of his examples, in a post called “JRuby Interop DSL in Scala”.

I ported his JRuby wrapper code to Python.



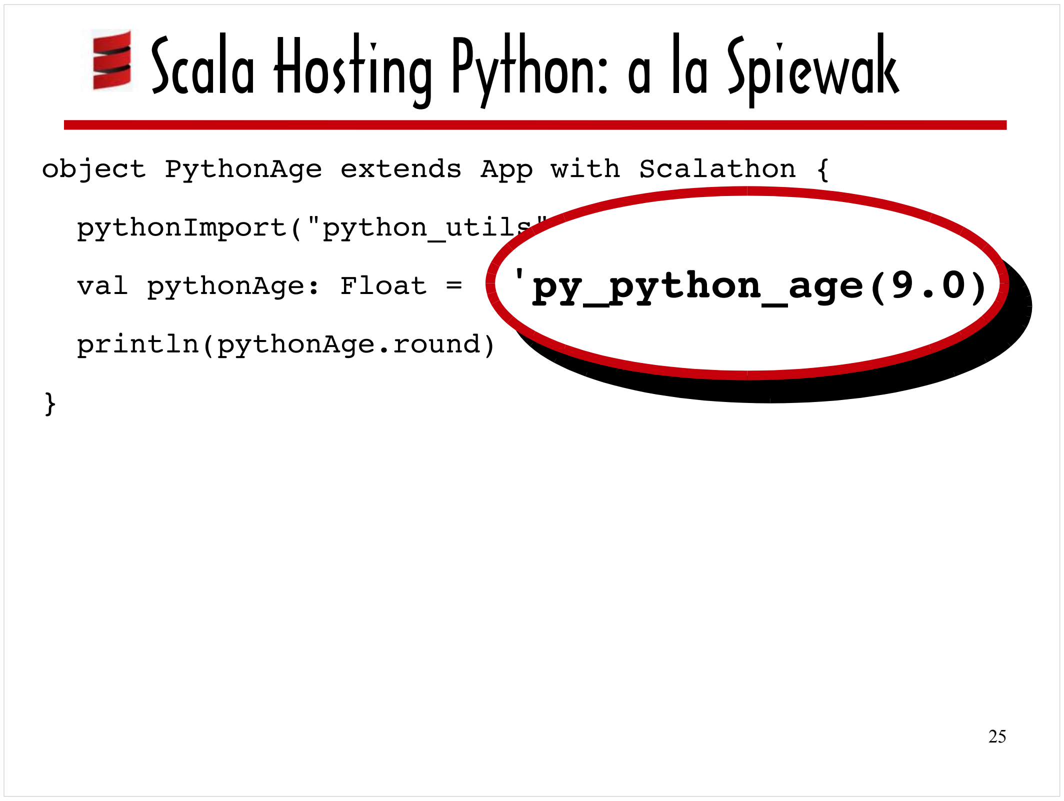
## Scala Hosting Python: a la Spiewak

```
object PythonAge extends App with Scalathon {
  python_import("python_utils")
  val pythonAge: Float = 'py_python_age(9.0)
  println(pythonAge.round)
}
```

Here's a Scala program that uses Daniel Spiewak's wrapping scheme to allow the Scala program to access the **`py_python_age`** function defined in my Python micro-library.

I'll be stepping through this code line by line, and then we'll walk through the code that makes this kind of wrapping possible.

You don't need to flip back to the beginning of the slides if you don't remember what the first version of this program looked like. I'll show you both in a split screen comparison shortly.



## Scala Hosting Python: a la Spiewak

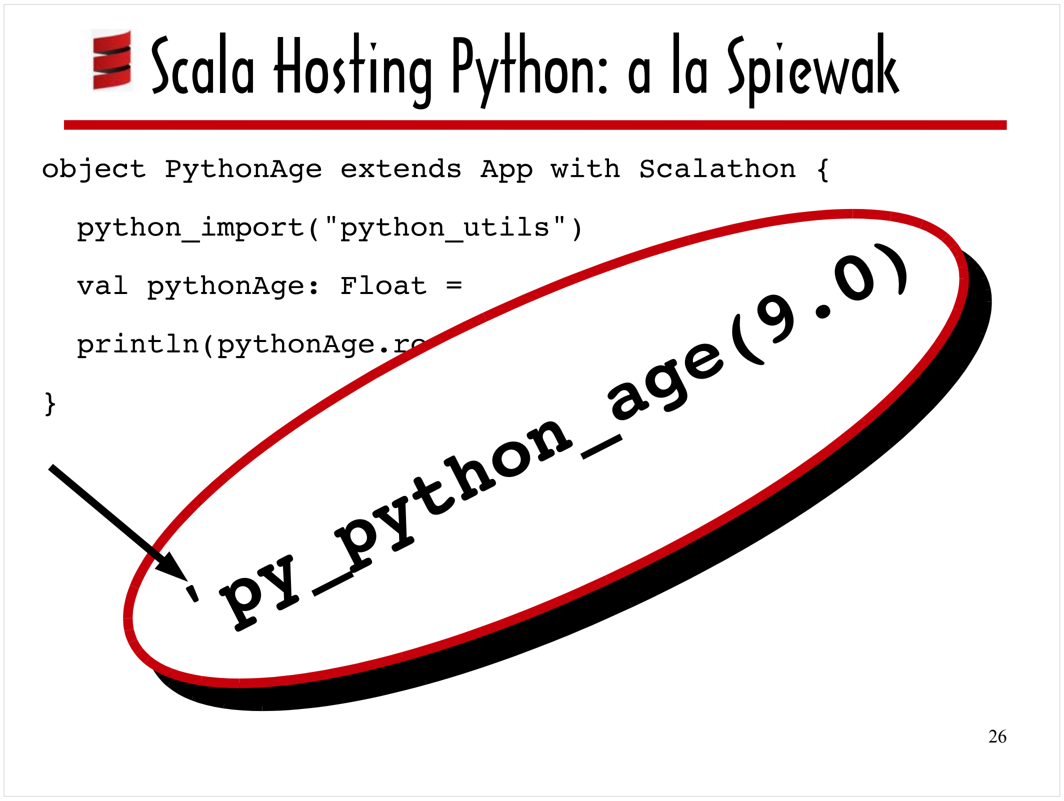
```
object PythonAge extends App with Scalathon {
  pythonImport("python_utils"
  val pythonAge: Float = 'py_python_age(9.0)
  println(pythonAge.round)
}
```

25

Here I've highlighted the call to the the Python micro-library's **`py_python_age`** function so you could get a good look at it.

As you can see, there's virtually no difference between the way this Python function is called and the way I would call a function written in Scala.

I say "virtually" before there is a difference. The difference is that the function name...



## Scala Hosting Python: a la Spiewak

```
object PythonAge extends App with Scalathon {
  python_import("python_utils")
  val pythonAge: Float =
  println(pythonAge.ro
}
```

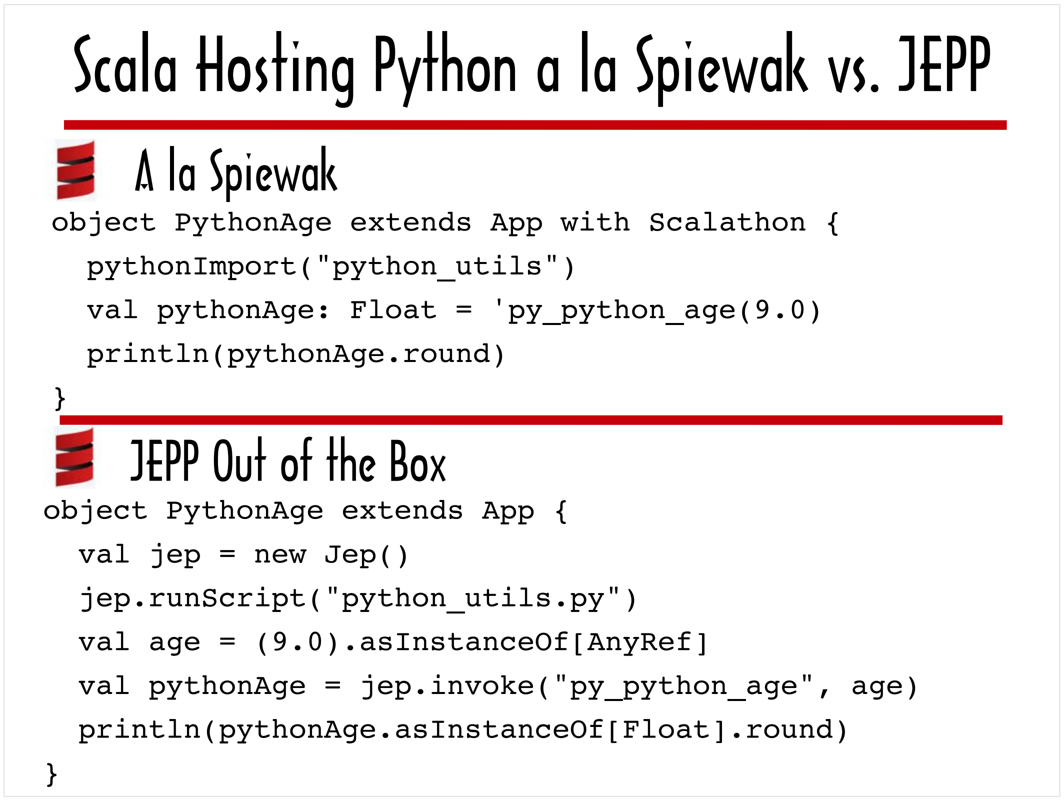
'py_python_age(9.0)

… is preceded by a single quote..

In Scala, preceding text with a single quote denotes a symbol. There is no Python equivalent to a Scala symbol. Symbols are interned. They are passed in argument lists as flags, and they can be used as keys in key/value pairs.

You don't see Scala symbols used conventionally in lieu of function names.

In a bit, I'll show you how Spiewak's idea of using symbols in this novel way is a key part of the wrapping mechanism he designed.



## Scala Hosting Python a la Spiewak vs. JEPP

### A la Spiewak

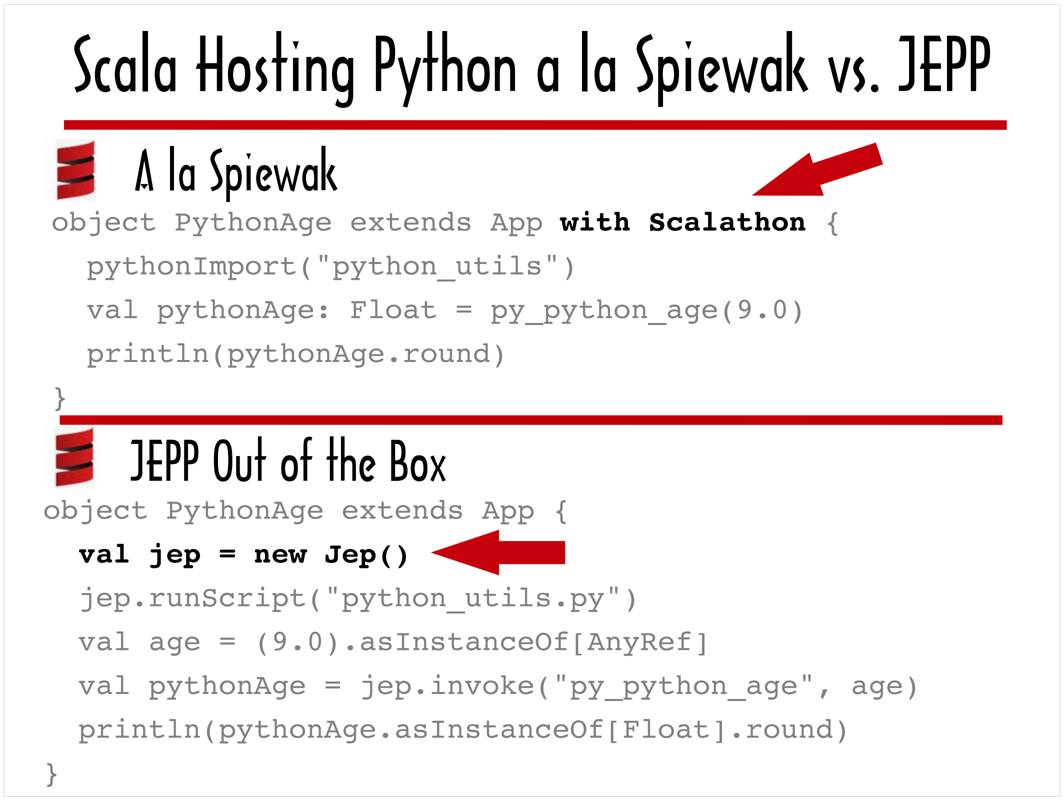
```
object PythonAge extends App with Scalathon {
  pythonImport("python_utils")
  val pythonAge: Float = 'py_python_age(9.0)
  println(pythonAge.round)
}
```

### JEPP Out of the Box

```
object PythonAge extends App {
  val jep = new Jep()
  jep.runScript("python_utils.py")
  val age = (9.0).asInstanceOf[AnyRef]
  val pythonAge = jep.invoke("py_python_age", age)
  println(pythonAge.asInstanceOf[Float].round)
}
```

Here's a first look at JEPP a la Spiewak (on the top half of the slide) vs. JEPP Out of the Box (on the bottom half of the slide).

The first thing I'd like to point out is that the Spiewak-inspired version is more compact. It does the same thing in fewer lines, and the lines are generally shorter.



## Scala Hosting Python a la Spiewak vs. JEPP

### A la Spiewak

```
object PythonAge extends App with Scalathon {
  pythonImport("python_utils")
  val pythonAge: Float = py_python_age(9.0)
  println(pythonAge.round)
}
```

### JEPP Out of the Box

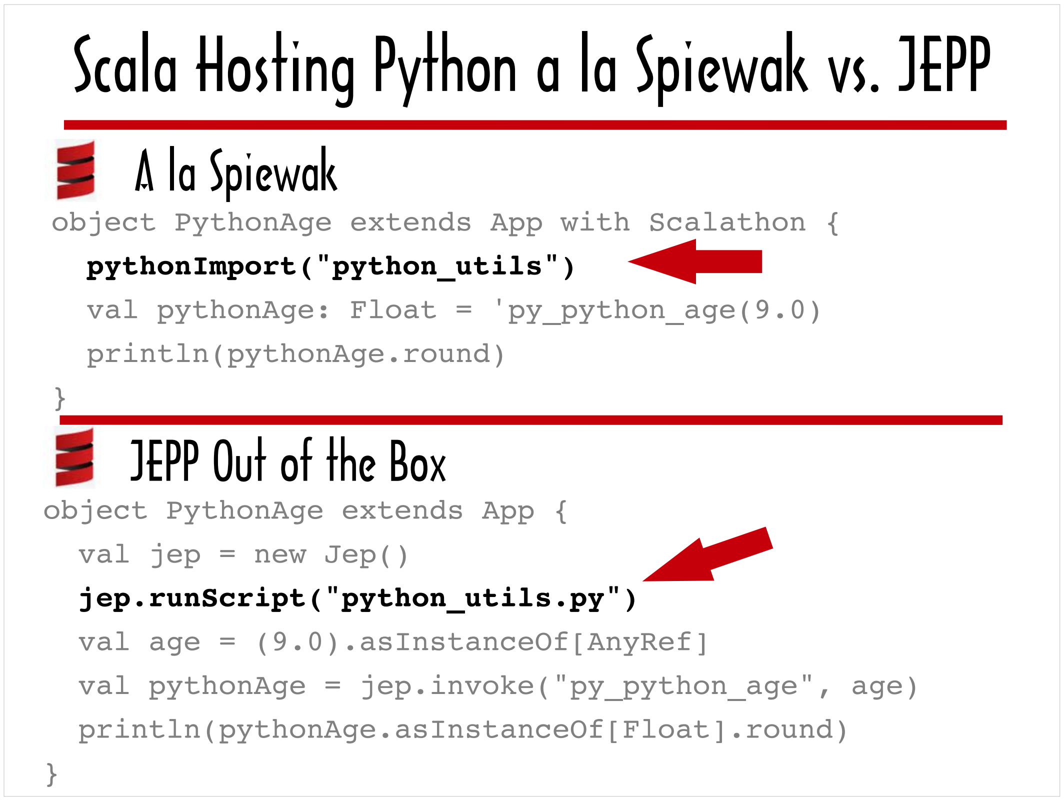
```
object PythonAge extends App {
  val jep = new Jep()
  jep.runScript("python_utils.py")
  val age = (9.0).asInstanceOf[AnyRef]
  val pythonAge = jep.invoke("py_python_age", age)
  println(pythonAge.asInstanceOf[Float].round)
}
```

**`Scalathon`**, in the phrase I've highlighted in top half of the slide (**`with Scalathon`**), is a Scala trait. A Scala trait is similar in some ways to an abstract base class in Python and in some ways to a Python module.

A trait can include declarations for both abstract and concrete methods. The **`with`** keyword is part of Scala's support for mixins and serves to add the **`Scalathon`** trait's concrete methods to the **`PythonAge`** definition. If the **`Scalathon`** trait included any abstract methods, **`PythonAge`** would need to provide implementations for them. Any code, including inner class definitions in the **`Scalathon`** trait, becomes part of **`PythonAge`**.

**`Scalathon`** is a trait that facilitates Python/Scala interop and that I modeled after Daniel Spiewak's Scala/JRuby interop DSL. I called it **`Scalathon`** because it's one of many possible ways to combine the names 'Scala' and 'Python'. As you will see, I use other combinations of these language names as module names in other examples.

I've paired the phrase **`with Scalathon`** with the line that instantiates a **`Jep`** instance on the bottom half of the slide because instantiating a **`Jep`** instance is one of the aspects of using JEPP that the **`Scalathon`** trait encapsulates.



## Scala Hosting Python a la Spiewak vs. JEPP

### A la Spiewak

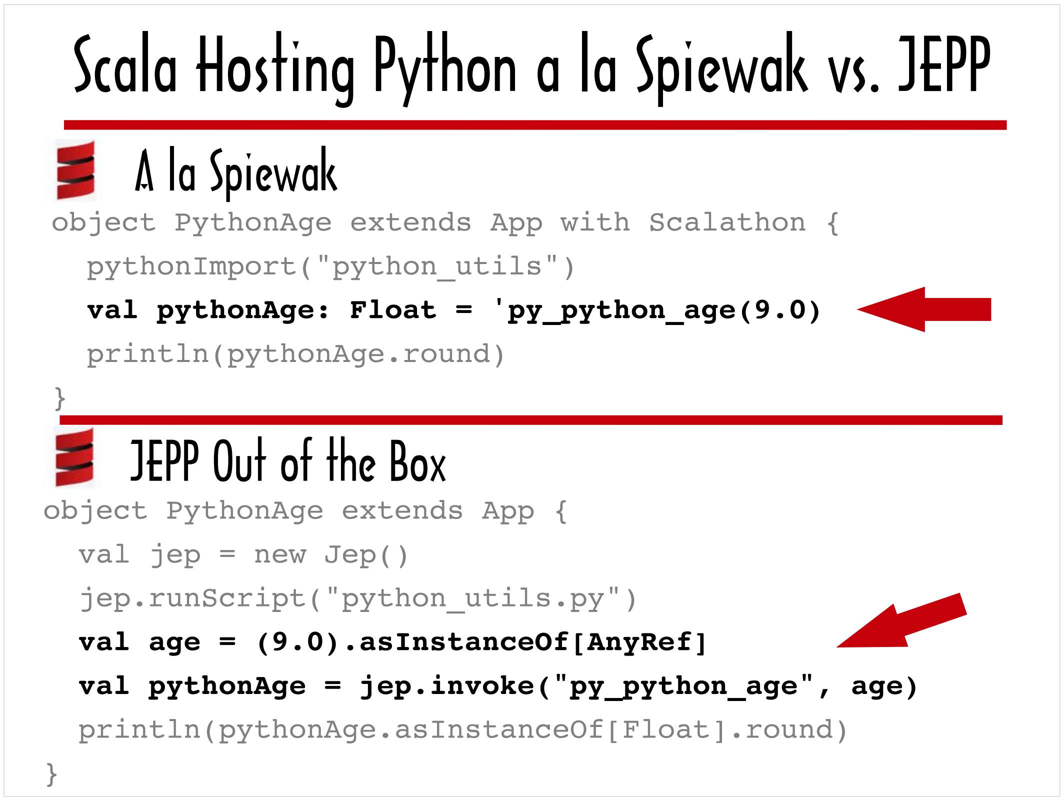
```
object PythonAge extends App with Scalathon {
  pythonImport("python_utils")
  val pythonAge: Float = 'py_python_age(9.0)
  println(pythonAge.round)
}
```

### JEPP Out of the Box

```
object PythonAge extends App {
  val jep = new Jep()
  jep.runScript("python_utils.py")
  val age = (9.0).asInstanceOf[AnyRef]
  val pythonAge = jep.invoke("py_python_age", age)
  println(pythonAge.asInstanceOf[Float].round)
}
```

Here, the method **`pythonImport`**, which I've highlighted in the Spiewak-inspired version, is a garden-variety wrapper (as opposed to the more exotic wrappers we'll be looking at) that is defined in the Scalathon trait. Basically, it wraps a call to **`jep.runScript`**. The purpose of the wrapper is to make running the Python script at least a little more like importing a Python module.

I've highlighted the **`jep.runScript`** call in the JEPP 'Out of the Box' version.



## Scala Hosting Python a la Spiewak vs. JEPP

### A la Spiewak

```
object PythonAge extends App with Scalathon {
  pythonImport("python_utils")
  val pythonAge: Float = 'py_python_age(9.0)
  println(pythonAge.round)
}
```

### JEPP Out of the Box

```
object PythonAge extends App {
  val jep = new Jep()
  jep.runScript("python_utils.py")
  val age = (9.0).asInstanceOf[AnyRef]
  val pythonAge = jep.invoke("py_python_age", age)
  println(pythonAge.asInstanceOf[Float].round)
}
```

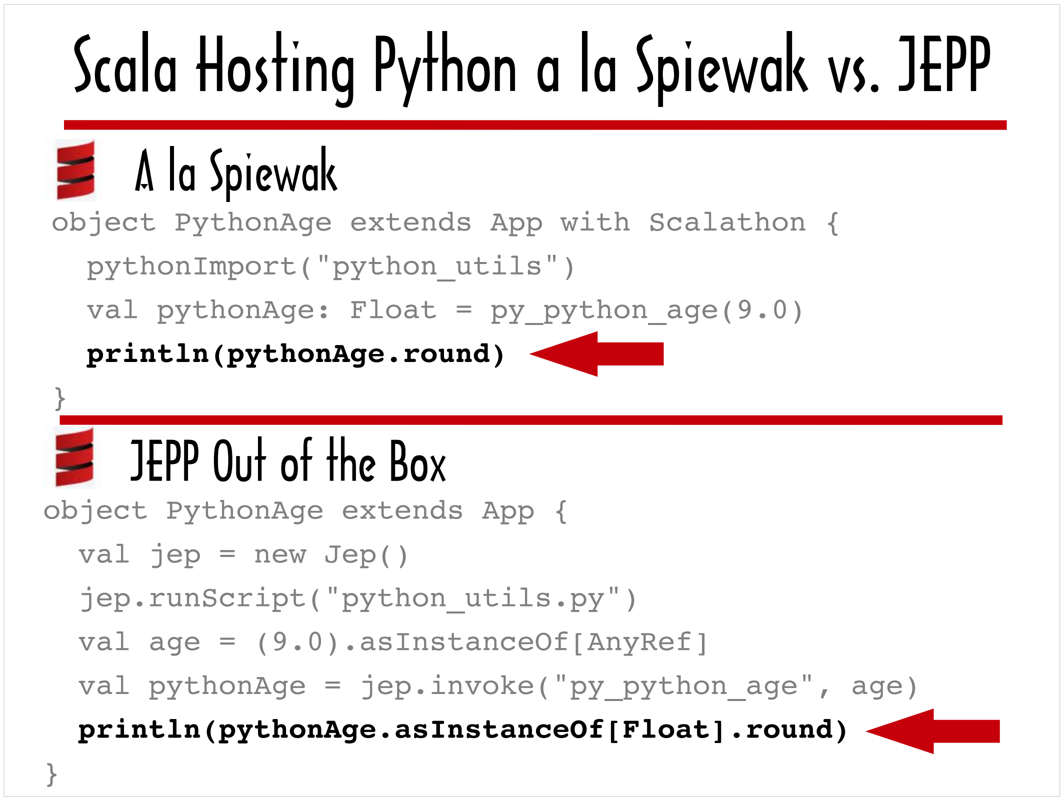
Here's where the Spiewak-inspired version shines.

This slide contrasts being able to call a function written in Python almost as if it were a function defined in Scala with needing to pass the function name and any arguments to **`Jep#invoke`**.

This slide also contrasts a more unobtrusive kind of typecasting in the Spiewak-inspired version with the  more heavy-handed **`asInstanceOf`** in the 'Out of the Box' version.

The Scala syntax used for typecasting in the 'A la Spiewak' version is something I have not shown you in a Scala program before. In Scala, you can specify the type for a variable by following it with a colon and the type.

I'll talk about the significance of using the type Float cast in the top of the slide vs, the type AnyRef cast on the bottom on the next slide.



## Scala Hosting Python a la Spiewak vs. JEPP

### A la Spiewak

```
object PythonAge extends App with Scalathon {
  pythonImport("python_utils")
  val pythonAge: Float = py_python_age(9.0)
  println(pythonAge.round)
}
```

### JEPP Out of the Box

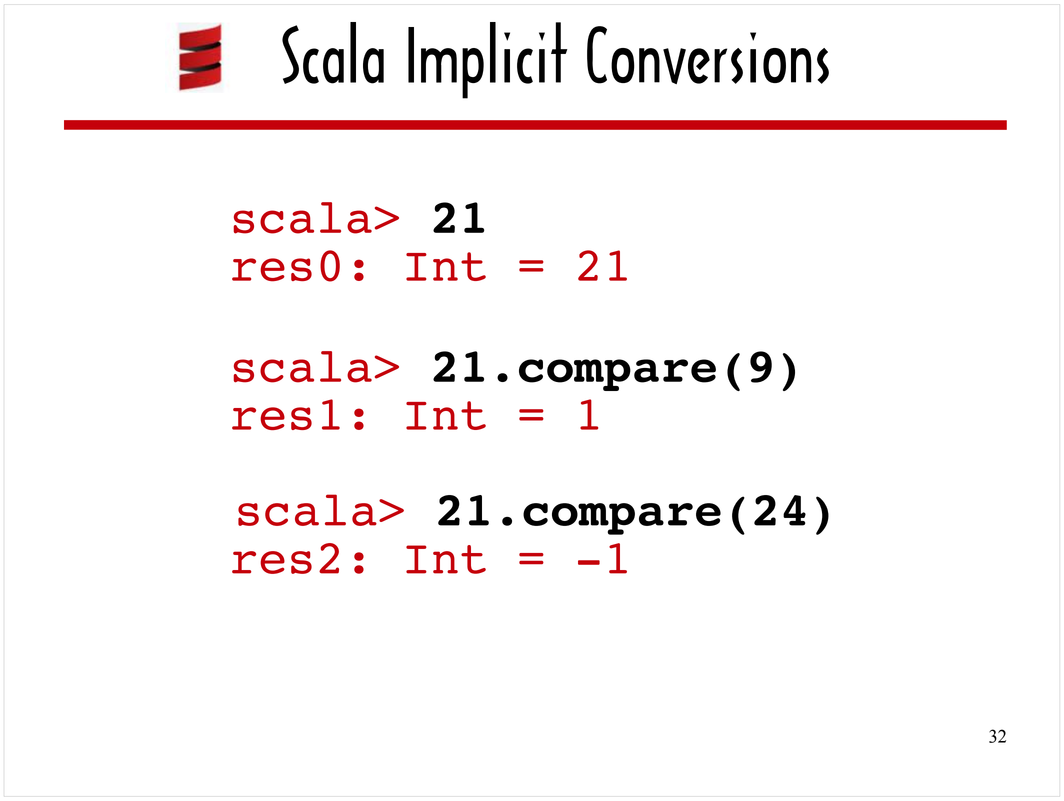
```
object PythonAge extends App {
  val jep = new Jep()
  jep.runScript("python_utils.py")
  val age = (9.0).asInstanceOf[AnyRef]
  val pythonAge = jep.invoke("py_python_age", age)
  println(pythonAge.asInstanceOf[Float].round)
}
```

This slide points out that no typecast is necessary in order to call **`round`** on the value returned from the wrapped call to **`py_python_age`** in the 'A la Spiewak' version, while the JEPP 'Out of the Box' version requires the heavy-handed **`asInstanceOf[Float]`** on the bottom.

You may well be wondering if the typecast on the bottom could be avoided if the **`age`** parameter was cast as a Float instead of the more general AnyRef. The answer is no. The signature of the Java-based JEPP method **`Jep#invoke`** is **`public Object invoke(String name, Object... args)`**, meaning that the return type is a Java Object. The return value of **`Jep#invoke`** is always going to be a generic Java Object and only methods defined on java.lang.Object can be invoked on it without a typecast in the calling code.

The reason that **`round`** can be called on **`pythonAge`** in the 'A La Spiewak' version without typecasting has to do with the way **`Jep#invoke`** is wrapped in the **`Scalathon`** trait, and we'll look at the source code for that wrapper shortly.

In order to understand how the **`Scalathon`** trait works, however, there are a couple of additional things you need to know about Scala.



## Scala Implicit Conversions

```
scala> 21
res0: Int = 21

scala> 21.compare(9)
res1: Int = 1

scala> 21.compare(24)
res2: Int = -1
```

32

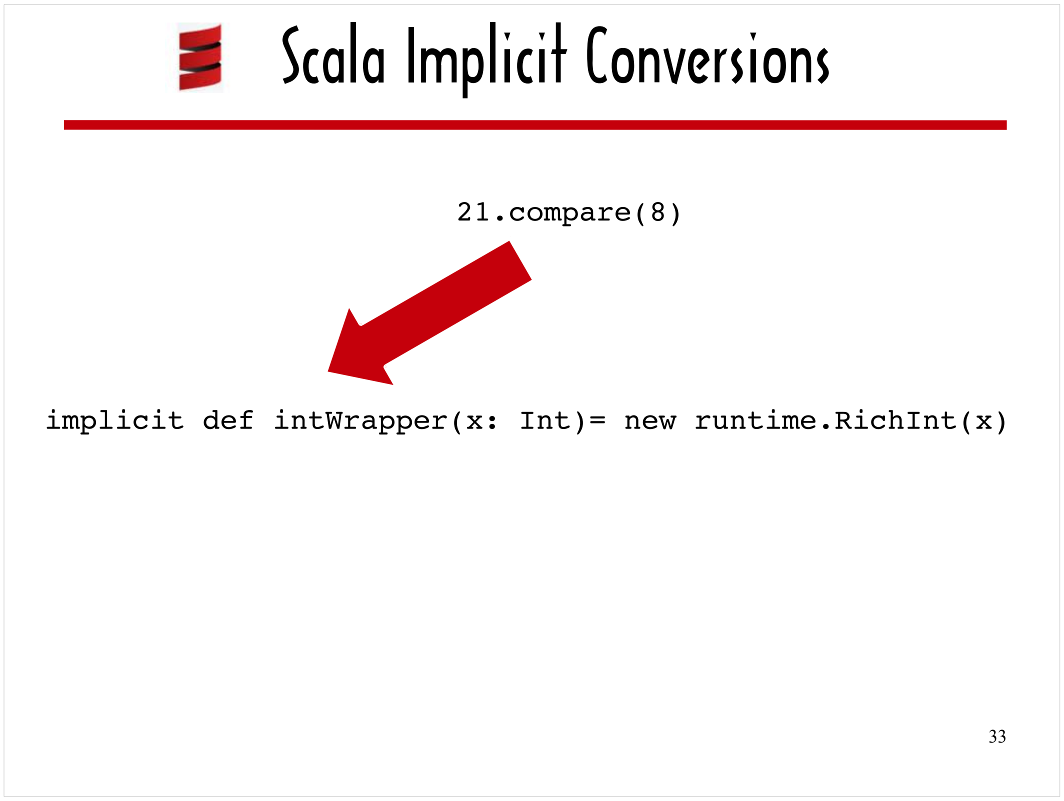
The first is how Scala's implicit conversions work.

This slide shows an implicit conversion in action. It captures a console session in which I invoked the interactive Scala prompt. What I entered is represented by black text, and the Scala prompt itself and any console output is colored red.

The Scala console reports the value and type of the expression on each line, so you can see that the `21`, entered on the first line is an Int. On the next 2 lines, I  call a method called `compare` on `21`. The `compare` method returns 1, -1 or 0 depending on whether the object `compare` is invoked on is greater than, less than, or equal to the number passed in.

It looks for all the world like the Int class supports the `compare` method, but in actual fact, it does not.

A Scala implicit conversion is responsible for this illusion. There are implicit conversions defined in the Scala source, and you can also declare your own.



## Scala Implicit Conversions

```
21.compare(8)
```

```
implicit def intWrapper(x: Int)= new runtime.RichInt(x)
```

33

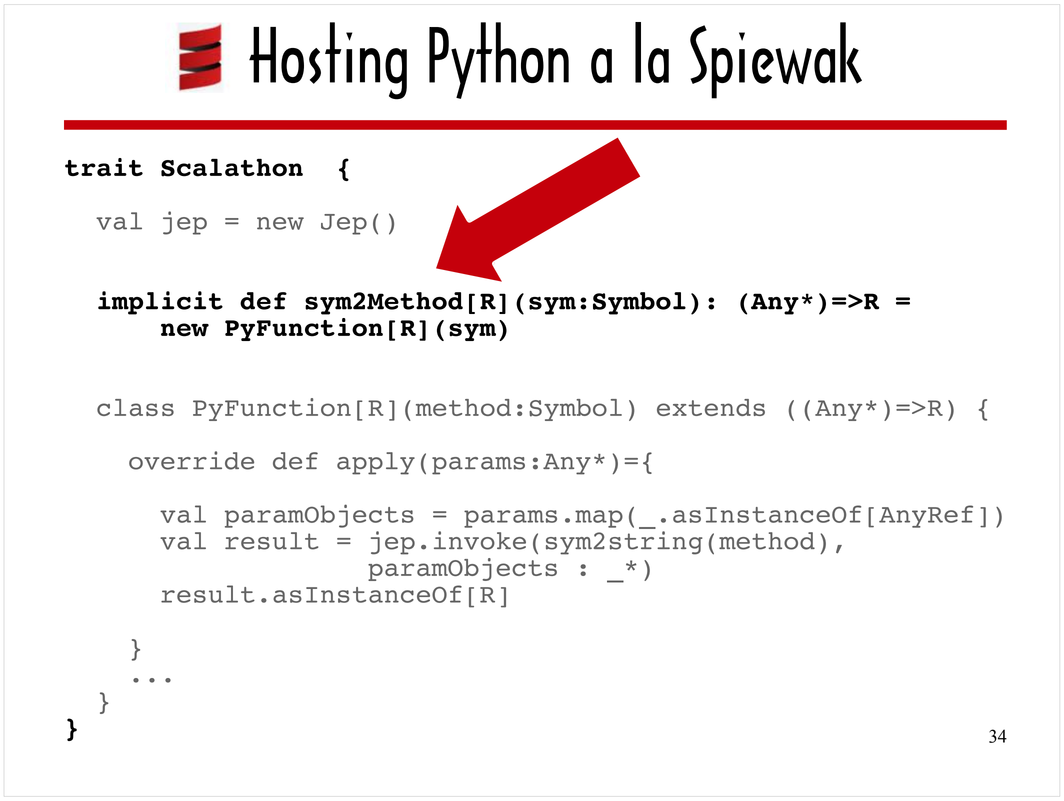
The arrow is pointing to the implicit conversion in the Scala core source code that makes it appear that it's possible to call **`compare`** on an Int.

When Scala encounters a call to a method that does not exist on the object it is called on, it checks to see if there are any implicit conversions defined for that kind of object.

This implicit conversion, called **`intWrapper`**, transforms an Int into a runtime.RichInt. It takes an Int argument called **`x`**, and passes it to the constructor for runtime.RichInt. The new runtime.RichInt is returned to the call site. RichInt does support **`compare`**, so Scala is able to call **`compare`** on the newly-minted RichInt without complaint.

The one additional thing you need to know about Scala before you look at the Scalathon source is that a method called **`apply`** in Scala is comparable to **`__call__`** in Python. When Scala encounters zero or more arguments flanked by parentheses following an object identifier, Scala invokes that object's **`apply`** method and passes the arguments between the parentheses to **`apply`**.

What do you think happens if the object does not support the **`apply`** method? If the object does not support the apply method, Scala looks for an implicit conversion, and executes the implicit conversion if one is found.



## Hosting Python a la Spiewak

```
trait Scalathon  {

  val jep = new Jep()

  implicit def sym2Method[R](sym:Symbol): (Any*)=>R =
     new PyFunction[R](sym)

  class PyFunction[R](method:Symbol) extends ((Any*)=>R) {

    override def apply(params:Any*)={

      val paramObjects = params.map(_.asInstanceOf[AnyRef])
      val result = jep.invoke(sym2string(method),
                   paramObjects : _*)
      result.asInstanceOf[R]

    }
    ...
  }
}
```

34

Here's a compacted version **Scalathon** source.

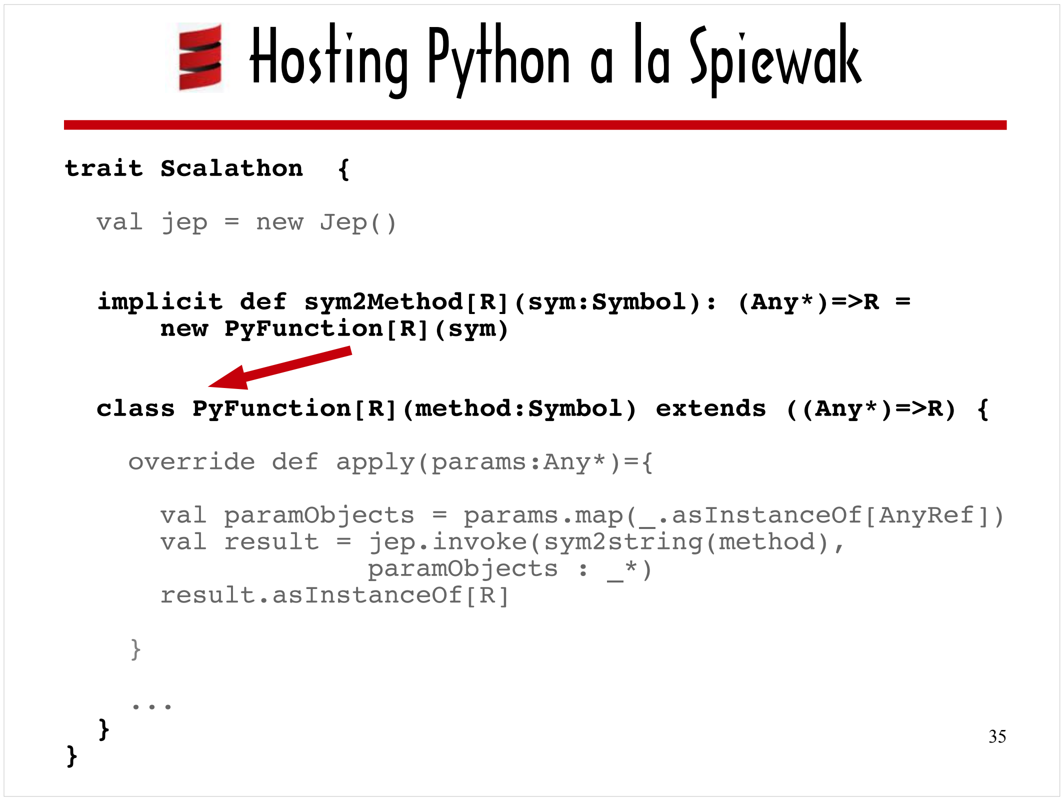
The first thing I want to point out is the implicit conversion defined for a Scala symbol.

Recall that in the Spiewak-inspired version, a symbol ('**py_python_age**) was used in lieu of a function identifier in the phrase **'py_python_age(9.0)**.

When Scala sees **(9.0)**, which is an argument flanked by parentheses, following the symbol **'py_python_age**, it tries to call **apply** on the symbol **'py_python_age**.

When Scala realizes that there is no **apply** method defined for the Symbol class, Scala looks to see if there are any implicit conversations defined for symbols -- and the implicit conversion defined in the **Scalathon** trait, which is highlighted above, gets triggered.

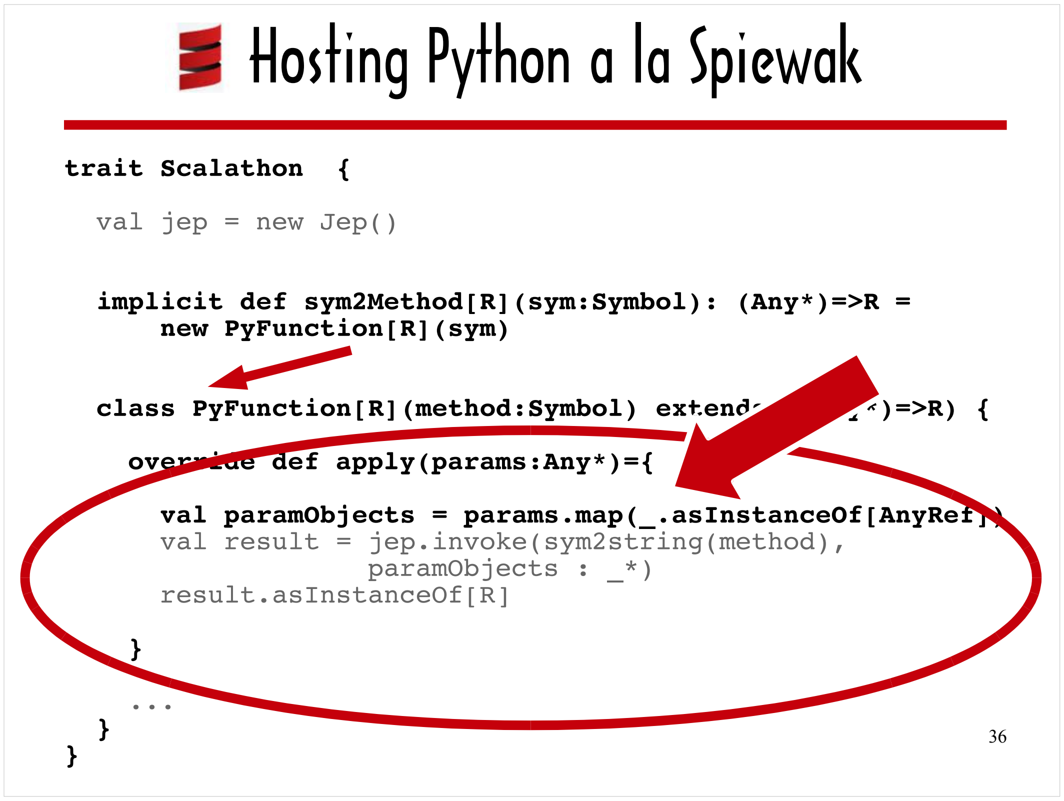
The implicit conversion in the **Scalathon** trait converts a symbol into a PyFunction object, just as the implicit conversion we looked at one the last slide converts an Int into a RichInt. The **sym2Function** implicit conversion passes the symbol (in this case **'py_python_age**) to the constructor for ...



## Hosting Python a la Spiewak

```
trait Scalathon  {

  val jep = new Jep()


  implicit def sym2Method[R](sym:Symbol): (Any*)=>R =
      new PyFunction[R](sym)


  class PyFunction[R](method:Symbol) extends ((Any*)=>R) {

    override def apply(params:Any*)={

      val paramObjects = params.map(_.asInstanceOf[AnyRef])
      val result = jep.invoke(sym2string(method),
                    paramObjects : _*)
      result.asInstanceOf[R]

    }

    ...
  }
}
```

… a class called PyFunction, and the class definition for the PyFunction class does ...



## Hosting Python a la Spiewak

```
trait Scalathon  {

  val jep = new Jep()

  implicit def sym2Method[R](sym:Symbol): (Any*)=>R =
      new PyFunction[R](sym)

  class PyFunction[R](method:Symbol) extend[illegible]*)=>R) {

    override def apply(params:Any*)={

      val paramObjects = params.map(_.asInstanceOf[AnyRef])
      val result = jep.invoke(sym2string(method),
                   paramObjects : _*)
      result.asInstanceOf[R]

    }

    ...
  }
}
```

36

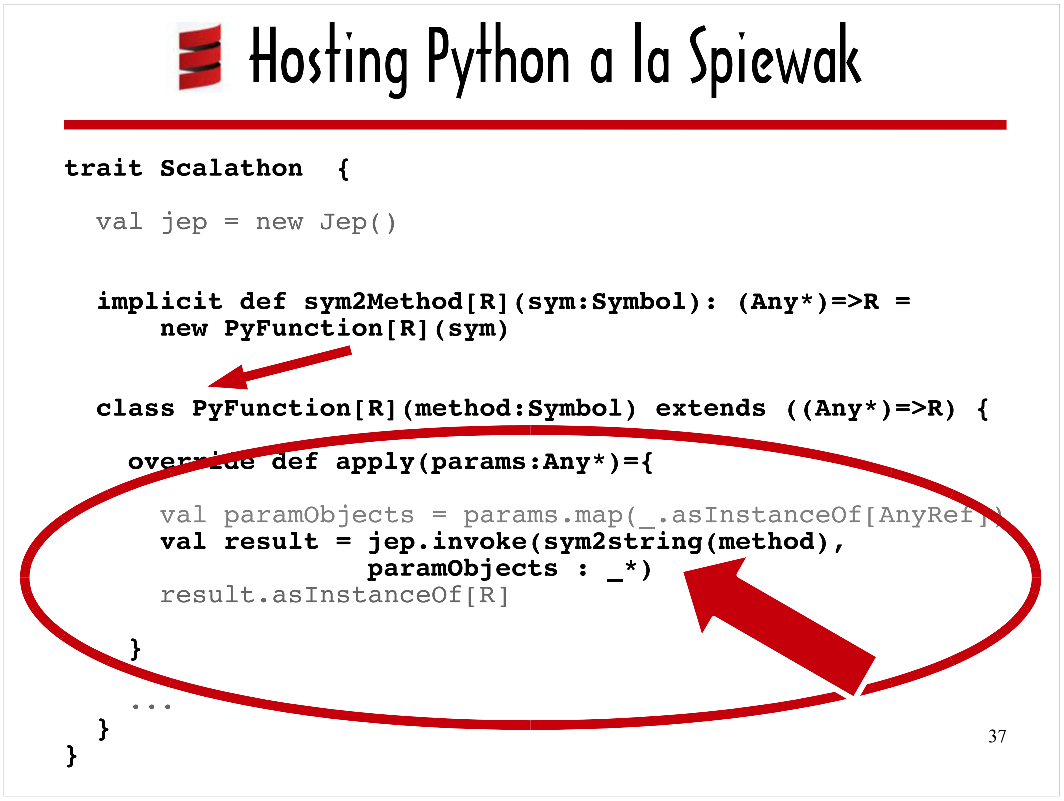
… include an implementation of **apply**.

The PyFunction class's **apply** implementation wraps a call to **Jep#invoke**.

Notice that the type specified for the **params** argument to **apply** is Any. We have discussed AnyRef, but not Any. One of the differences between Java and Scala is that in Scala, **9.0** is considered an object, but in Java, **9.0** is considered a primitive. A method that expects Any arguments can take **9.0**, but **9.0** needs to be converted to an AnyRef before it can passed to a method that expects AnyRef arguments.

The line highlighted in this slide takes care of converting the arguments to Java objects by calling **asInstanceOf[AnyRef]** on each one. Recall that Jep#invoke expects its arguments (that represent arguments to the Python function) to be instances of java.lang.Object.

Like Python's **map**, Scala's **map** takes a function parameter and creates a new collection comprised of the results of applying the passed-in function to each element. The phrase _.**asInstanceOf[AnyRef]** is a syntactic shortcut representing an anonymous function, with the underscore representing the function parameter.

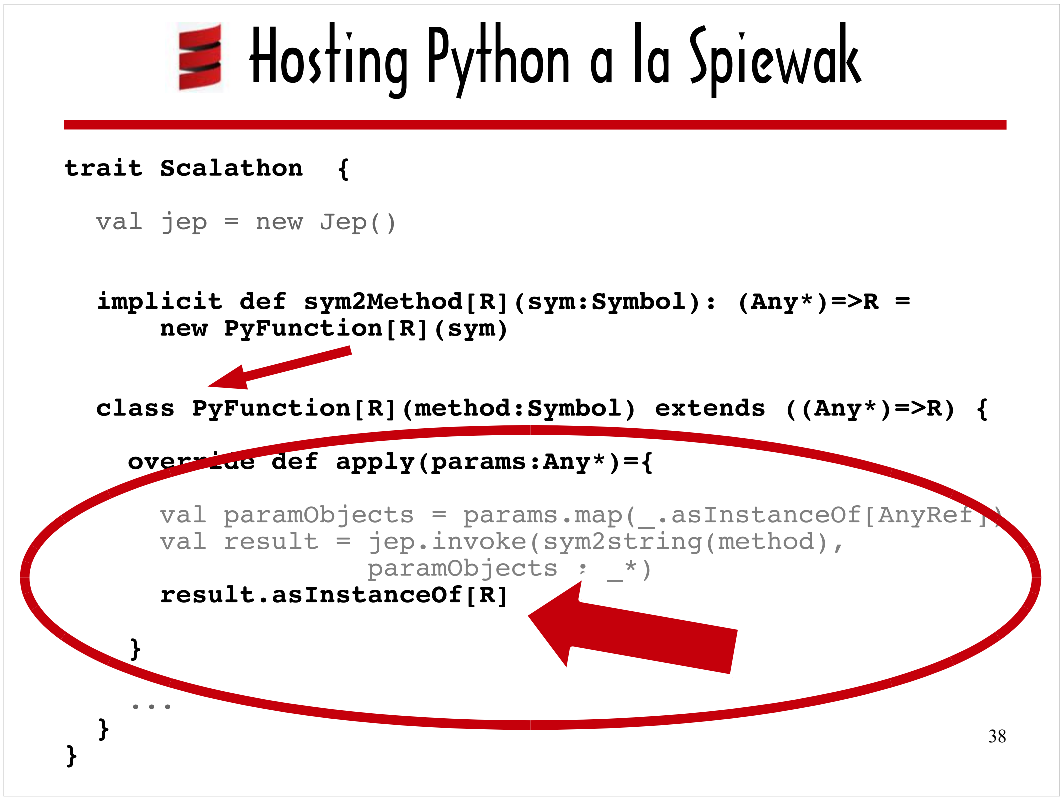


## Hosting Python a la Spiewak

```
trait Scalathon  {

  val jep = new Jep()

  implicit def sym2Method[R](sym:Symbol): (Any*)=>R =
      new PyFunction[R](sym)

  class PyFunction[R](method:Symbol) extends ((Any*)=>R) {

    override def apply(params:Any*)={

      val paramObjects = params.map(_.asInstanceOf[AnyRef])
      val result = jep.invoke(sym2string(method),
                   paramObjects : _*)
      result.asInstanceOf[R]

    }

    ...
  }
}
```

37

Here's where the name of the Python method and a variable number of arguments are actually passed to **`Jep#invoke`**. I have inserted ellipses in place of the **`sym2string`** code, which simply converts the  symbol to text and strips off the single quote.



## Hosting Python a la Spiewak

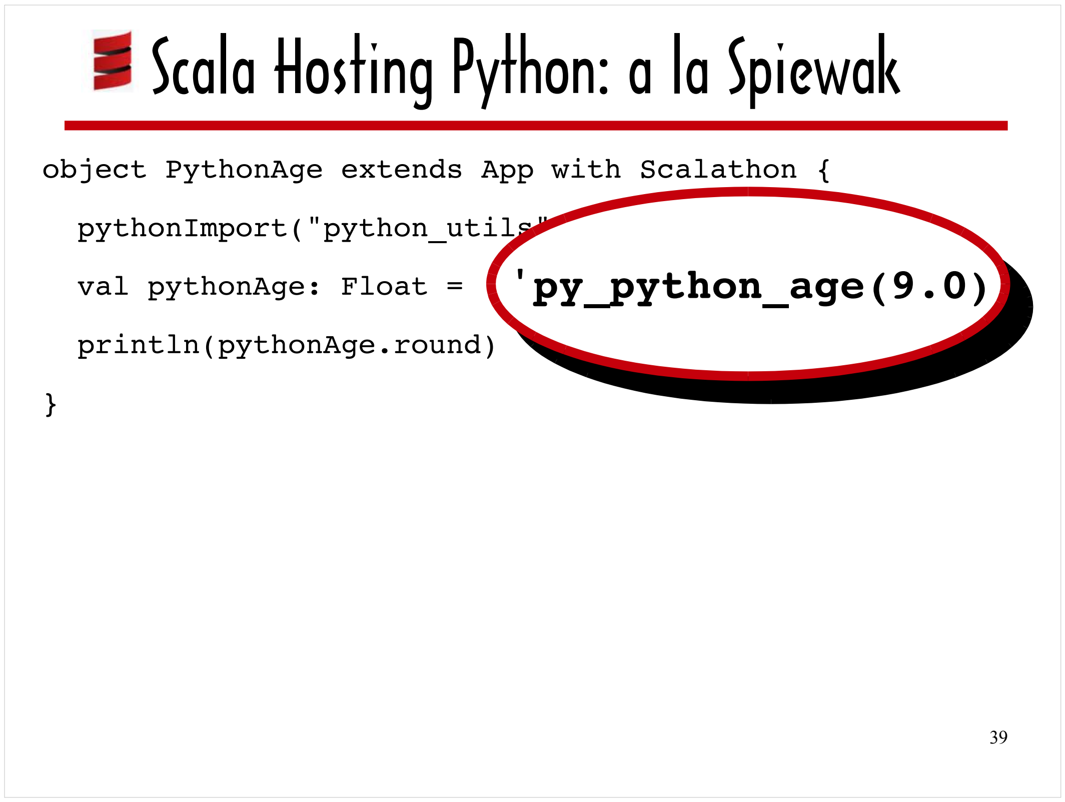
```
trait Scalathon  {

  val jep = new Jep()


  implicit def sym2Method[R](sym:Symbol): (Any*)=>R =
      new PyFunction[R](sym)


  class PyFunction[R](method:Symbol) extends ((Any*)=>R) {

    override def apply(params:Any*)={

      val paramObjects = params.map(_.asInstanceOf[AnyRef])
      val result = jep.invoke(sym2string(method),
                   paramObjects : _*)
      result.asInstanceOf[R]

    }

    ...
  }
}
```

38

This last line of **apply** is what makes it possible to call **round** in the Spiewak -inspired sample code without needing to do any typecasting.

How does the Scalathon trait know how to cast **result** (the return value of **Jep#invoke**)?

Brackets denote a parameterized type in Scala. By virtue of Scala's type inference system, the value of **[R]** is set to Float when the variable populated by the return value of **'py_python_age(9.0)** is cast as a Float as follows:
 **val pythonAge: Float**.

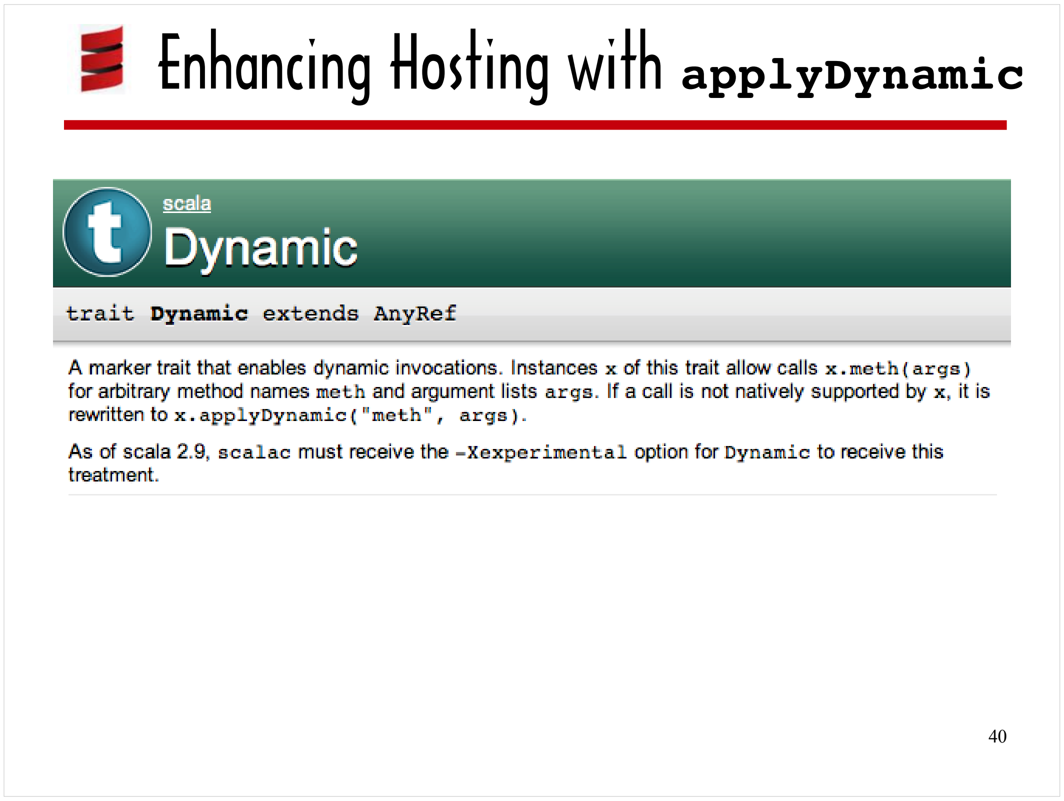


## Scala Hosting Python: a la Spiewak

```
object PythonAge extends App with Scalathon {
  pythonImport("python_utils"
  val pythonAge: Float = 'py_python_age(9.0)
  println(pythonAge.round)
}
```

39

Now that we've stepped through the source for the **`Scalathon`** trait, I wanted to give you another look at how to use the Spiewak-inspired JEPP wrapper.



## Enhancing Hosting with **`applyDynamic`**

scala

Dynamic

`trait Dynamic extends AnyRef`

A marker trait that enables dynamic invocations. Instances `x` of this trait allow calls `x.meth(args)` for arbitrary method names `meth` and argument lists `args`. If a call is not natively supported by `x`, it is rewritten to `x.applyDynamic("meth", args)`.

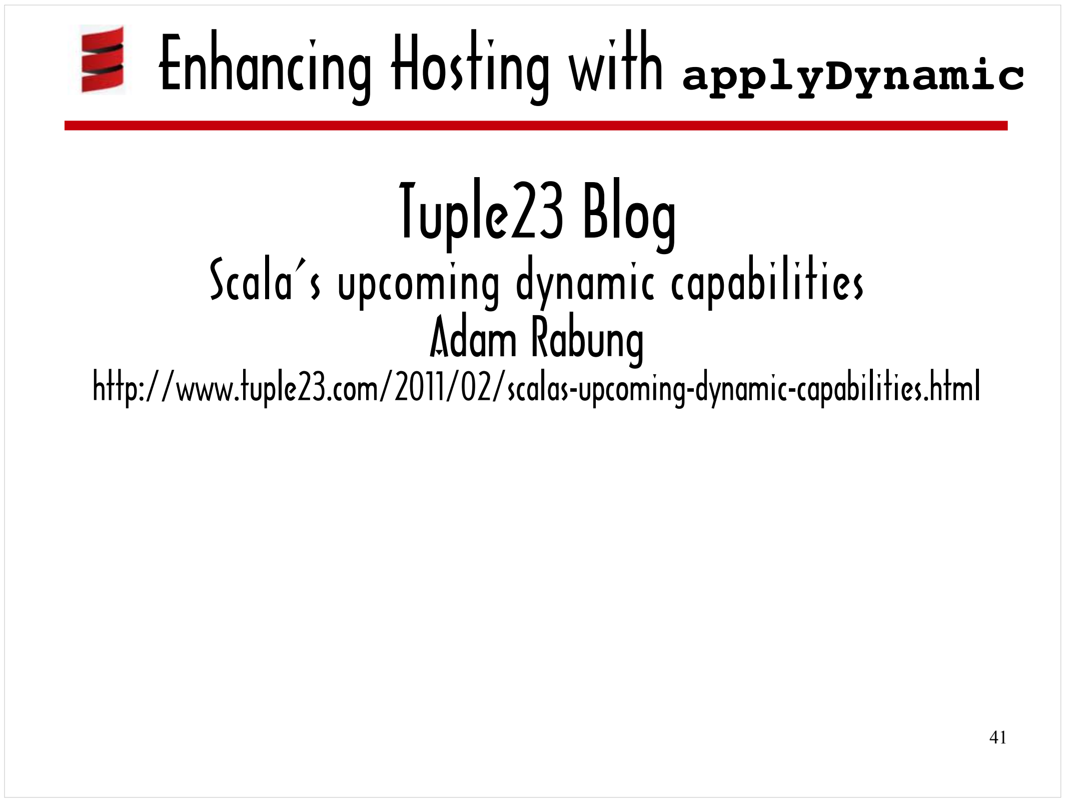
As of scala 2.9, `scalac` must receive the `-Xexperimental` option for `Dynamic` to receive this treatment.

In this next series of slides, we're going to look at a different way of wrapping JEPP code.

This next JEPP wrapper is built around an experimental Scala feature: the **`Dynamic`** trait, which supports the **`applyDynamic`** method.

The **`applyDynamic`** method is comparable to Python's **`__getattr__`**.

The ScalaDoc for the Dynamic explains that for an object **`x`** that mixes in the Dynamic trait and a call to **`x.meth (args)`** that "is not natively supported by **`x`**, it is rewritten to **`x.applyDynamic("meth", args)`**."



## Enhancing Hosting with `applyDynamic`

Tuple23 Blog

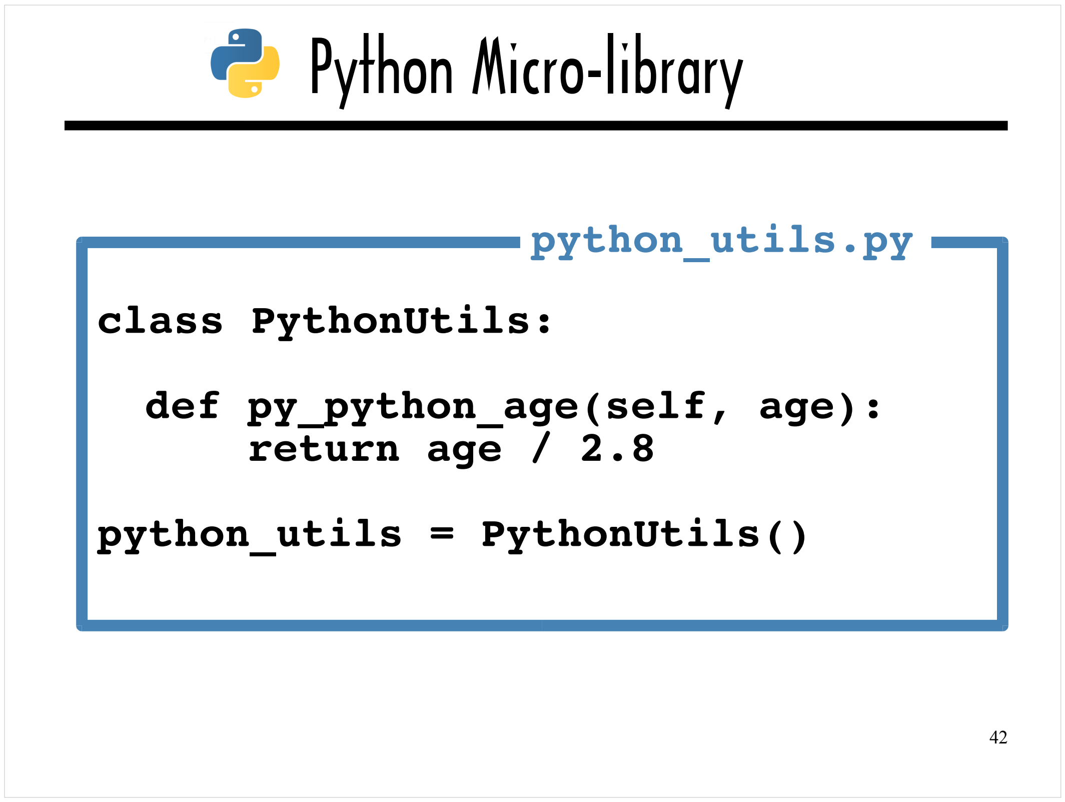
Scala's upcoming dynamic capabilities

Adam Rabung

http://www.tuple23.com/2011/02/scalas-upcoming-dynamic-capabilities.html

When I first learned about **`applyDynamic`**, I had an idea about how it could be used for wrapping JEPP. I couldn't get it to work until I read Adam Rabung's blog post "Scala's upcoming dynamic capabilities" which shows how **`applyDynamic`** can be used to integrate Scala with JRuby.

So, hat tip to Adam Rabung.



## Python Micro-library

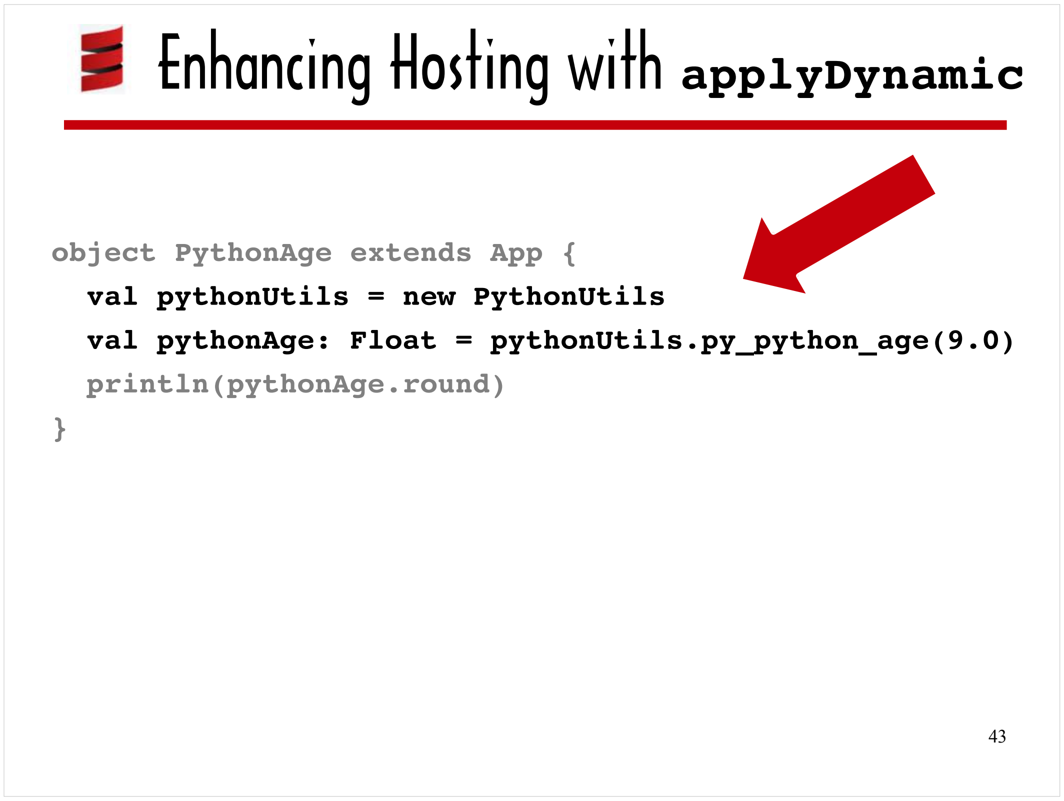
python_utils.py

```
class PythonUtils:

  def py_python_age(self, age):
      return age / 2.8

python_utils = PythonUtils()
```

42

For this **applyDynamic**-based wrapper, I modified **python_utils.py** by Adding a class definition for the PythonUtils class, making **py_python_age** An instance method for the PythonUtils class, instantiating an instance of PythonUtils, and assigning it to a variable called **python_utils**.

The cool thing about the **applyDynamic**-based JEPP wrapper is that it makes it appear that when Scala is the host language and Python is the guest, you can instantiate and call methods on an instance of a class defined in Python code exactly as if it were a class defined natively in Scala.



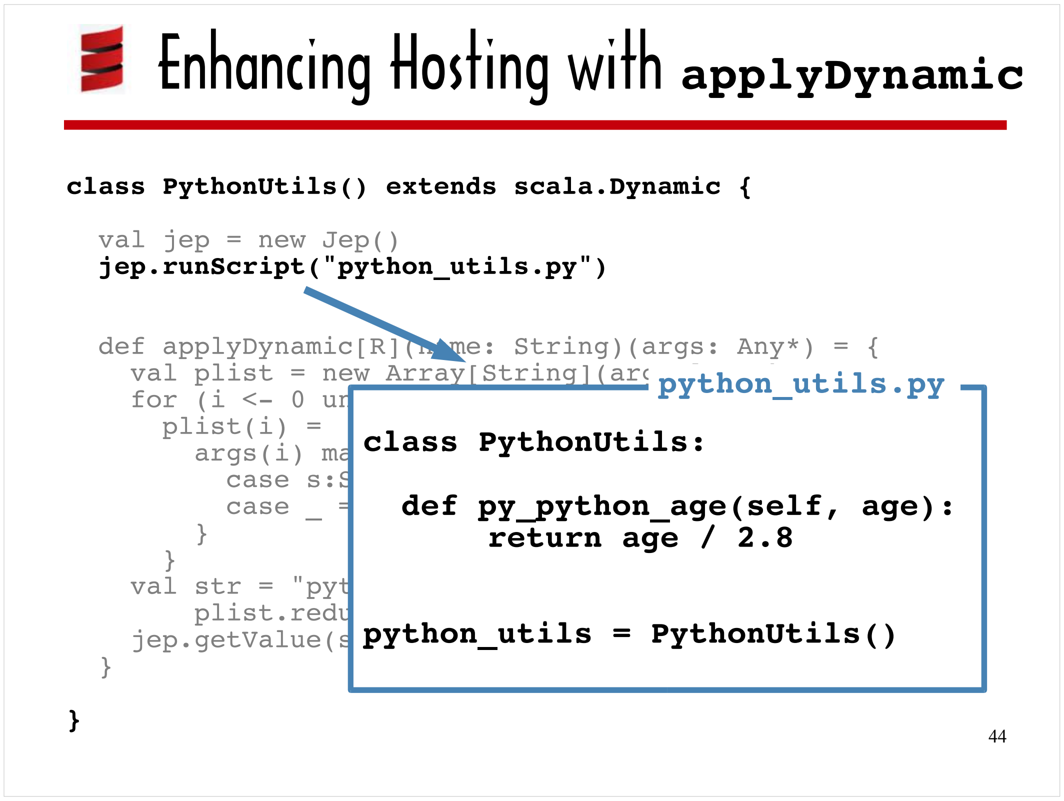
## Enhancing Hosting with **`applyDynamic`**

```
object PythonAge extends App {
  val pythonUtils = new PythonUtils
  val pythonAge: Float = pythonUtils.py_python_age(9.0)
  println(pythonAge.round)
}
```

This example shows how the **`applyDynamic`**-based JEPP wrapper can be used in a Scala program.

The highlighted lines of code serve to invoke **`py_python_age`** on an instance of the Python class PythonUtils defined in **`python_utils.py`**. The syntax is identical to the syntax for calling methods on an instance of a Scala class!

In the next slide we'll start looking at the source code that is responsible for this unobtrusive Scala\Python integration.



## Enhancing Hosting with `applyDynamic`

```
class PythonUtils() extends scala.Dynamic {

  val jep = new Jep()
  jep.runScript("python_utils.py")


  def applyDynamic[R](name: String)(args: Any*) = {
    val plist = new Array[String](arc
    for (i <- 0 ur
      plist(i) =
        args(i) ma
          case s:S
          case _ =
        }
      }
    val str = "pyt
        plist.redu
    jep.getValue(s
  }

}
```

python_utils.py

```
class PythonUtils:

  def py_python_age(self, age):
        return age / 2.8


python_utils = PythonUtils()
```

44

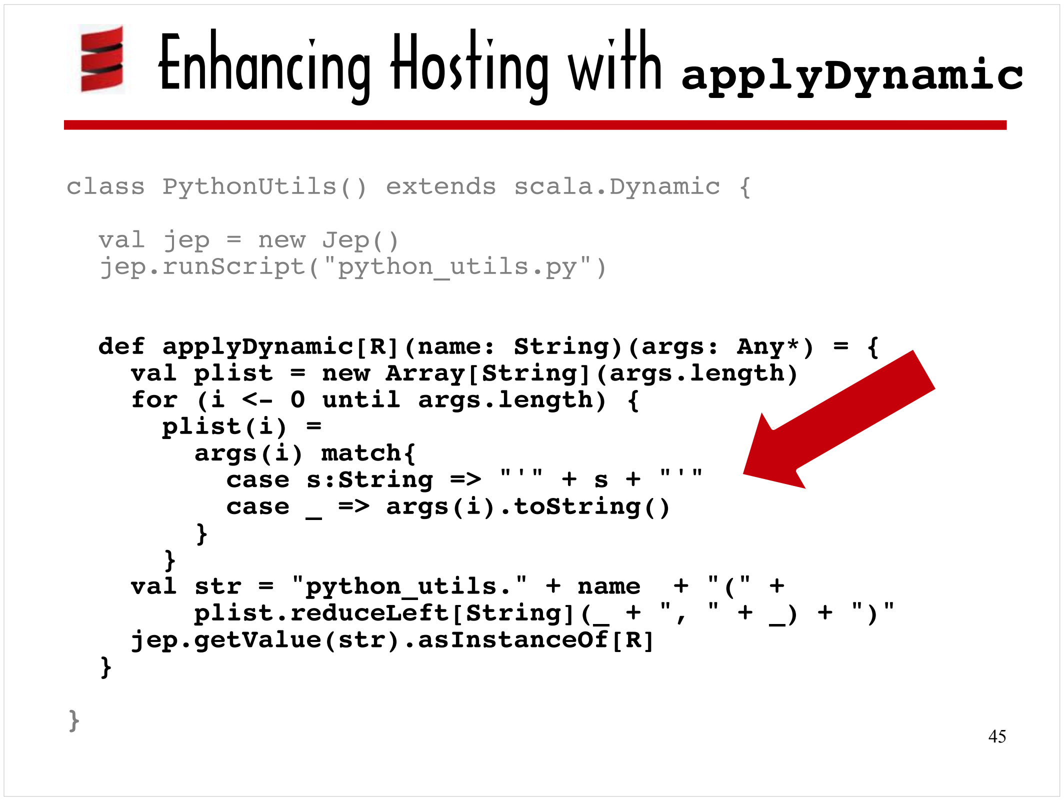
Using the **`applyDyamic`**-based wrapper involves defining a Scala proxy class that corresponds to the Python class you wish to access from a Scala program and that mixes in the **`Dynamic`** trait. The Scala proxy class and the Python class should have the same name. In this case, it's PythonUtils.

The Scala proxy class should encapsulate the **`Jep#runScript`** call that interprets the Python file that contains the Python class definition and the statement that assigns an instance of the Python class to a variable.

The Scala proxy class must mix in the **`Dynamic`** trait and implement **`applyDyamic`**. You can't get a good look at how **`applyDyamic`** is implemented in this slide, because it's grayed out and blocked by the **`python_utils.py`** source I'm displaying for reference. We'll focus on the **`applyDynamic`** source in the next slide.

But I do want to point out that there are no ellipses is this slide inserted as placeholders. You're looking at the sum total of the Scala proxy class source! Note that there is no definition for a method called **`py_python_age`** anywhere in sight in the Scala source.

Because **`applyDynamic`** is implemented to support calls to arbitrary methods on the Python-based PythonUtils class, the Scala proxy class definition does not need to have individual wrapper methods for each of the Python-based instance methods.



## Enhancing Hosting with `applyDynamic`

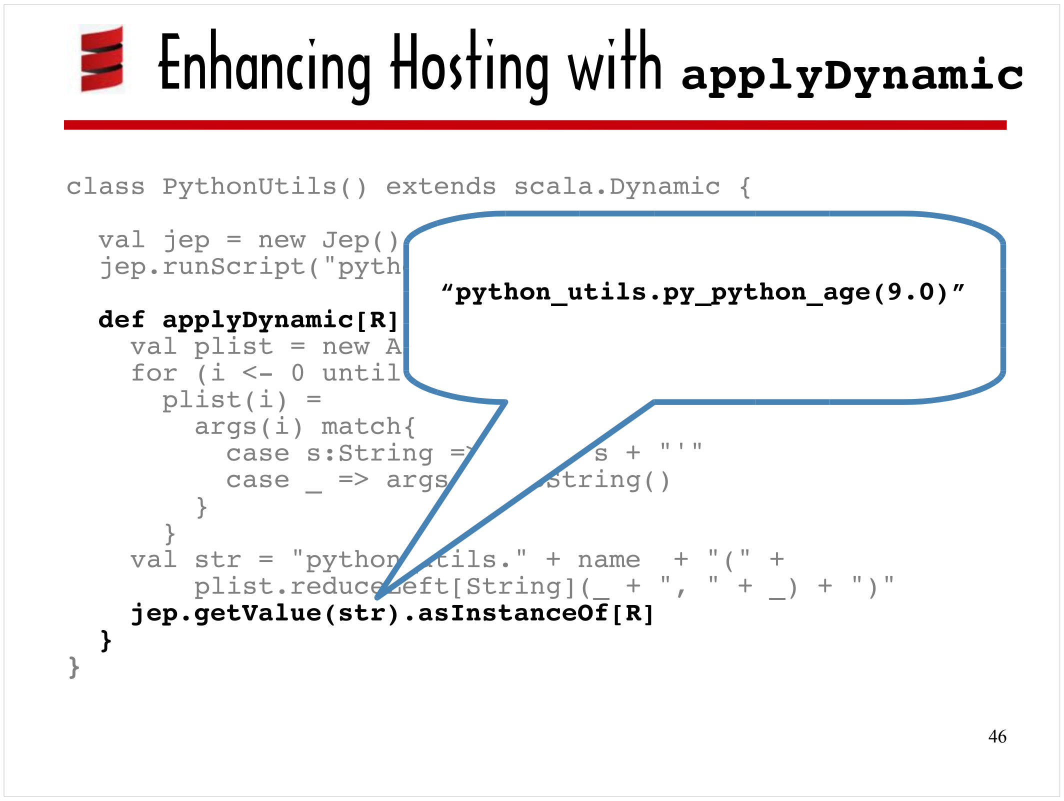
```
class PythonUtils() extends scala.Dynamic {

  val jep = new Jep()
  jep.runScript("python_utils.py")


  def applyDynamic[R](name: String)(args: Any*) = {
    val plist = new Array[String](args.length)
    for (i <- 0 until args.length) {
      plist(i) =
        args(i) match{
          case s:String => "'" + s + "'"
          case _ => args(i).toString()
        }
      }
    val str = "python_utils." + name  + "(" +
        plist.reduceLeft[String](_ + ", " + _) + ")"
    jep.getValue(str).asInstanceOf[R]
  }

}
```

45

When a method that is not defined on the Scala proxy class is called on an instance of that class, the method name is passed to **`applyDynamic`** via the **`name`** parameter and the arguments are passed in via the **`args`** parameter.

The **`applyDynamic`** implementation in this Scala that acts as a proxy for PythonUtils wraps **`Jep#getValue`**, which is similar to **`Jep#invoke`**, but instead of taking a function name and a variable list of arguments as parameters, it takes a string representing arbitrary Python source code as its only argument. It passes this string to the Python/C API function **`PyRun_String`**, which executes it.

The **`applyDynamic`** implementation tacks the supplied method name onto a source code string that begins with “python_utils.” and then converts each argument to a string, separates the arguments by commas and tacks on the argument string flanked by parentheses...



## Enhancing Hosting with `applyDynamic`

```
class PythonUtils() extends scala.Dynamic {

  val jep = new Jep()
  jep.runScript("pyth

  def applyDynamic[R]
    val plist = new A
    for (i <- 0 until
      plist(i) =
        args(i) match{
          case s:String =>          s + "'"
          case _ => args        String()
        }
      }
    val str = "python  tils." + name  + "(" +
        plist.reduce eft[String](_ + ", " + _) + ")"
    jep.getValue(str).asInstanceOf[R]
  }
}
```

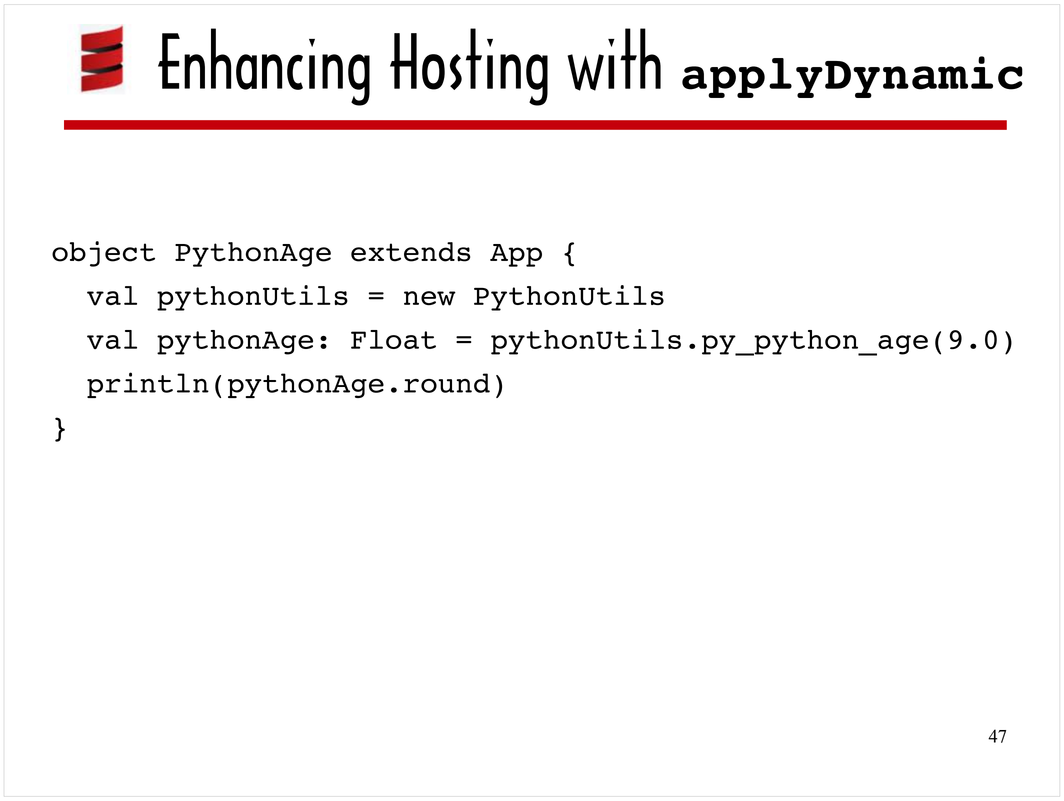
46

… to wind up with the source code in the balloon outlined in blue. This source string calls **`python_age`** on the instance of the Python-based PythonUtils class assigned to the **`python_utils`** variable in **`python_utils.py`**:
**`“python_utils.py_python_age(9.0).”`**

Because **`Jep#getValue`** can execute arbitrary Python source code, new methods added to the Python-based PythonUtils class definition can be called from within Scala programs without any changes to the Scala-based proxy class.

As in the Spiewak-inspired wrapper, Scala's type inference policy is responsible for determining the value of the **`applyDynamic`** type parameter **`[R]`** – so that the return value of **`applyDynamic`** can be typecast based on the type specified for the **`pythonAg`**e variable, which, in this case, is Float.

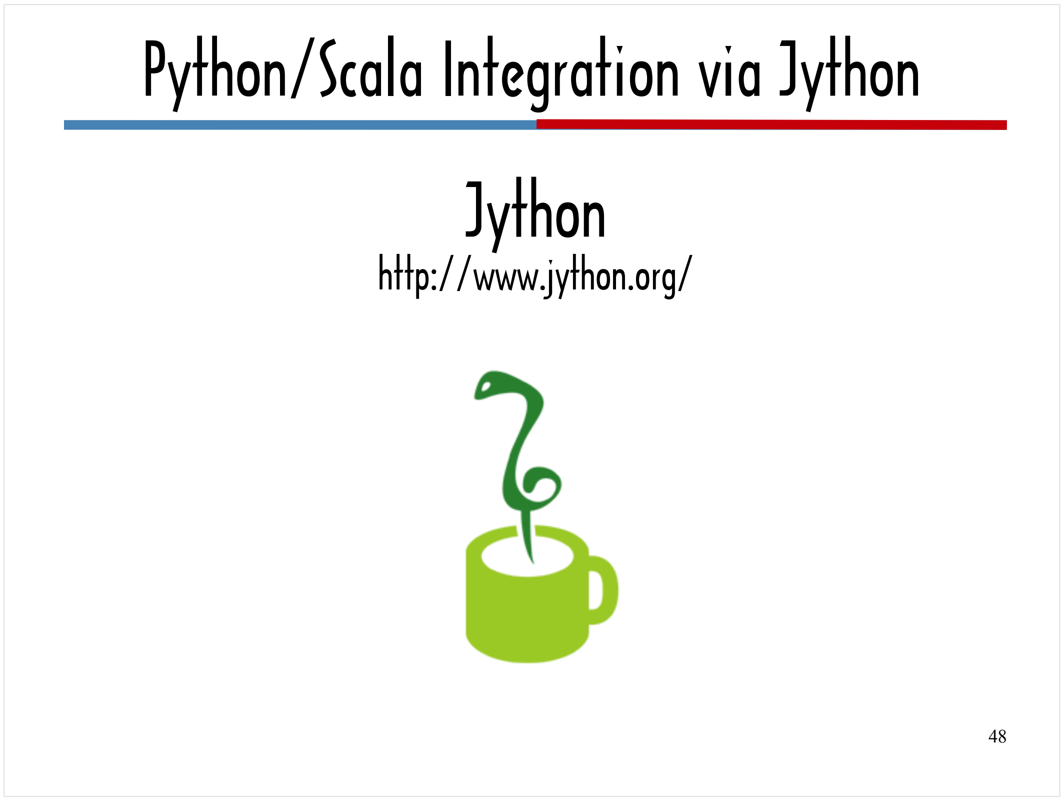
:



## Enhancing Hosting with **`applyDynamic`**

```
object PythonAge extends App {
  val pythonUtils = new PythonUtils
  val pythonAge: Float = pythonUtils.py_python_age(9.0)
  println(pythonAge.round)
}
```

Before moving on, I want to zoom back out of the **`applyDynamic`**-based wrapper source for another look at the sample Scala code that uses the wrapper.



## Python/Scala Integration via Jython

Jython

http://www.jython.org/

48

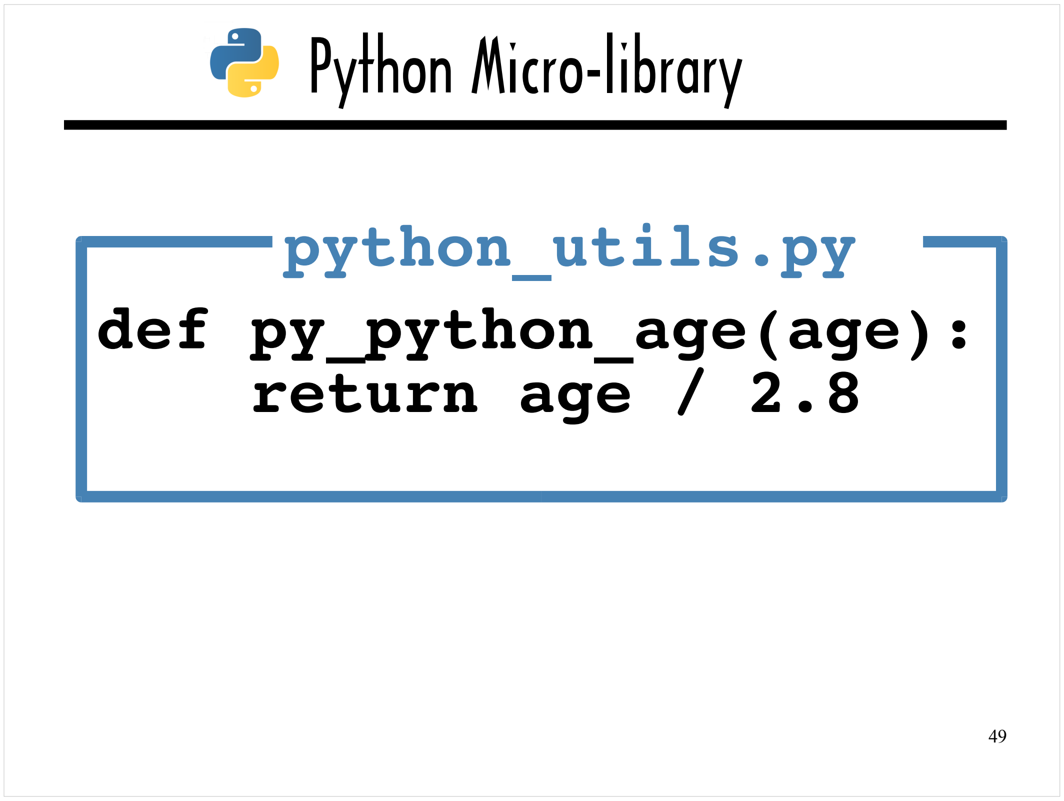
An approach to Python/Scala integration that is completely different than JEPP-based Python/Scala interop is using Jython to interpret the Python syntax.

Jython is an implementation of Python that runs on the JVM.

There are several different ways to embed Jython in Scala. The one I'm going to show you uses the JSR 223-compliant scripting engine that Jython supports.

A JSR (Java Service Request) is comparable to a PEP in the Python world.

JSR 223 defines a common hosting API for scripting languages that run on the JVM, enabling developers to support multiple scripting languages with the same host source code.

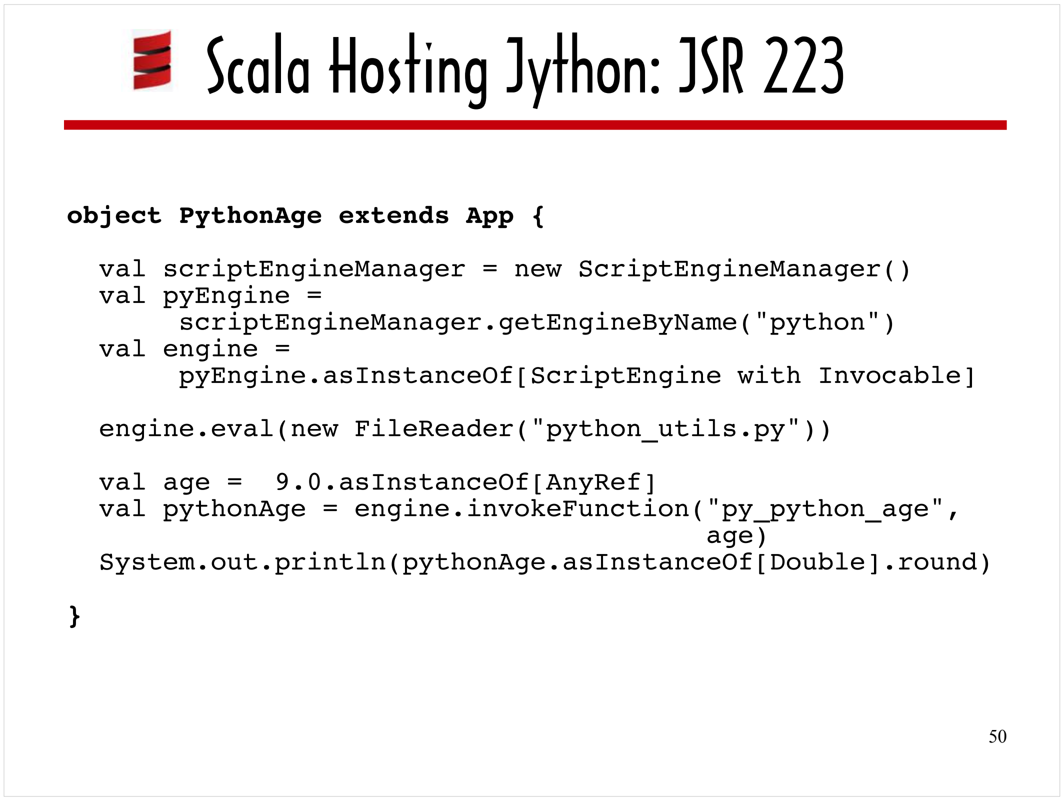


## Python Micro-library

python_utils.py

```
def py_python_age(age):
    return age / 2.8
```

For this next code sample, we'll use the original version of **`python_utils.py`**, in which **`py_python_age`** is a function, not an instance method defined for the PythonUtils class.

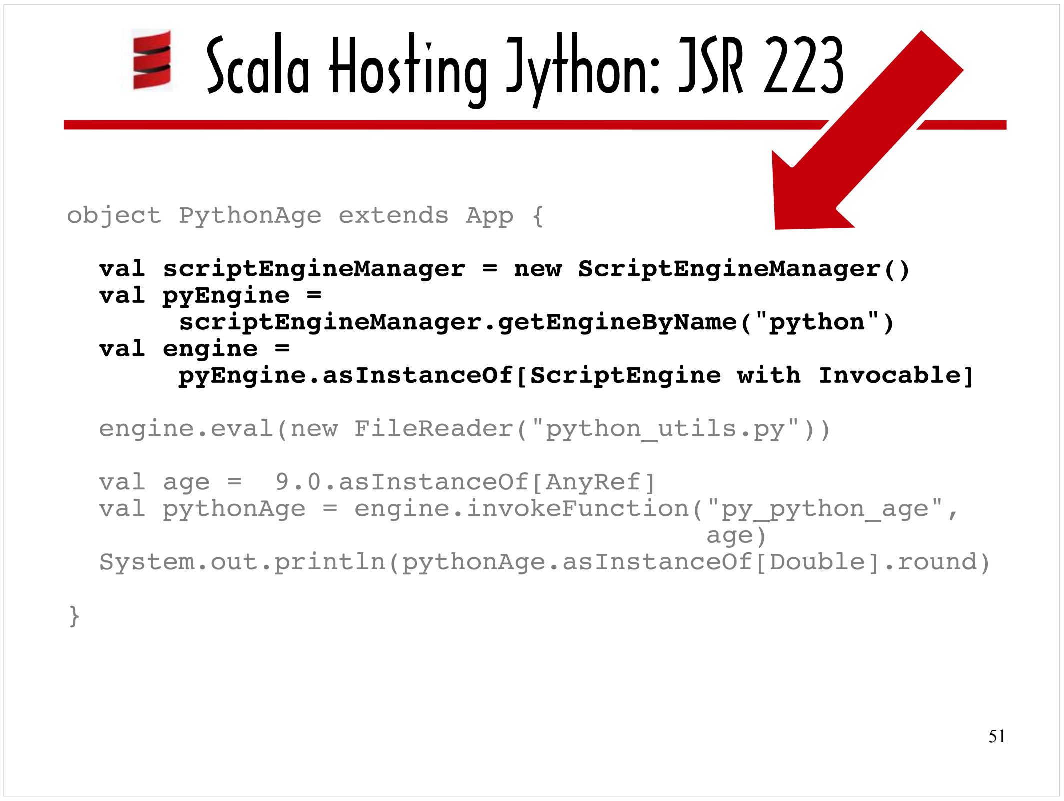


## Scala Hosting Jython: JSR 223

```
object PythonAge extends App {

  val scriptEngineManager = new ScriptEngineManager()
  val pyEngine =
       scriptEngineManager.getEngineByName("python")
  val engine =
       pyEngine.asInstanceOf[ScriptEngine with Invocable]

  engine.eval(new FileReader("python_utils.py"))

  val age =  9.0.asInstanceOf[AnyRef]
  val pythonAge = engine.invokeFunction("py_python_age",
                                        age)
  System.out.println(pythonAge.asInstanceOf[Double].round)

}
```

Here's a first look at sample Scala code that uses Jython's JSR 223-compliant scripting engine.

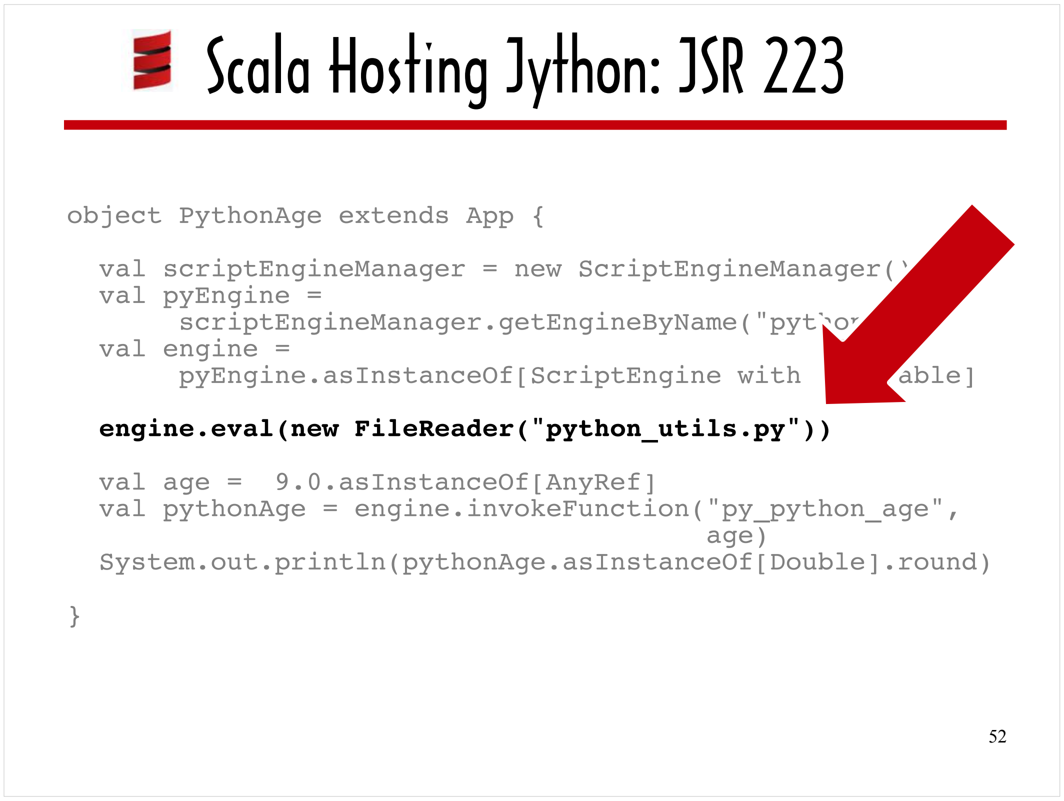
I'll walk through this code in the next series of slides.



## Scala Hosting Jython: JSR 223

```
object PythonAge extends App {

  val scriptEngineManager = new ScriptEngineManager()
  val pyEngine =
       scriptEngineManager.getEngineByName("python")
  val engine =
       pyEngine.asInstanceOf[ScriptEngine with Invocable]

  engine.eval(new FileReader("python_utils.py"))

  val age =  9.0.asInstanceOf[AnyRef]
  val pythonAge = engine.invokeFunction("py_python_age",
                                        age)
  System.out.println(pythonAge.asInstanceOf[Double].round)

}
```

First, the Jython scripting engine is instantiated and configured. The JSR 223 scripting engine API offers methods for invoking functions written in the guest language, as well as methods for executing arbitrary expressions written in the guest languages.

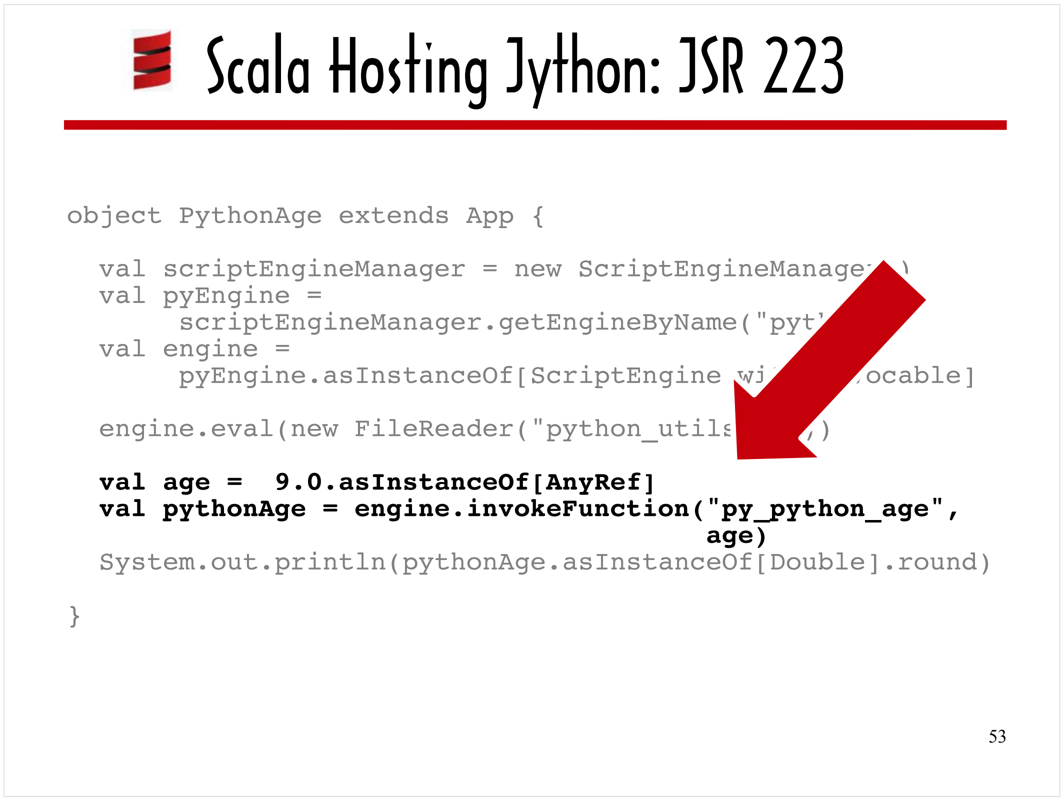


## Scala Hosting Jython: JSR 223

```
object PythonAge extends App {

  val scriptEngineManager = new ScriptEngineManager(
  val pyEngine =
       scriptEngineManager.getEngineByName("pyt or
  val engine =
       pyEngine.asInstanceOf[ScriptEngine with      able]

  engine.eval(new FileReader("python_utils.py"))

  val age =  9.0.asInstanceOf[AnyRef]
  val pythonAge = engine.invokeFunction("py_python_age",
                                        age)
  System.out.println(pythonAge.asInstanceOf[Double].round)

}
```

52

**`ScriptEngine#eval`** executes the code in the script file wrapped by its Reader argument, which is a **`FileReader`** (a subclass of Reader and a Convenience Class that facilitates processing character streams).



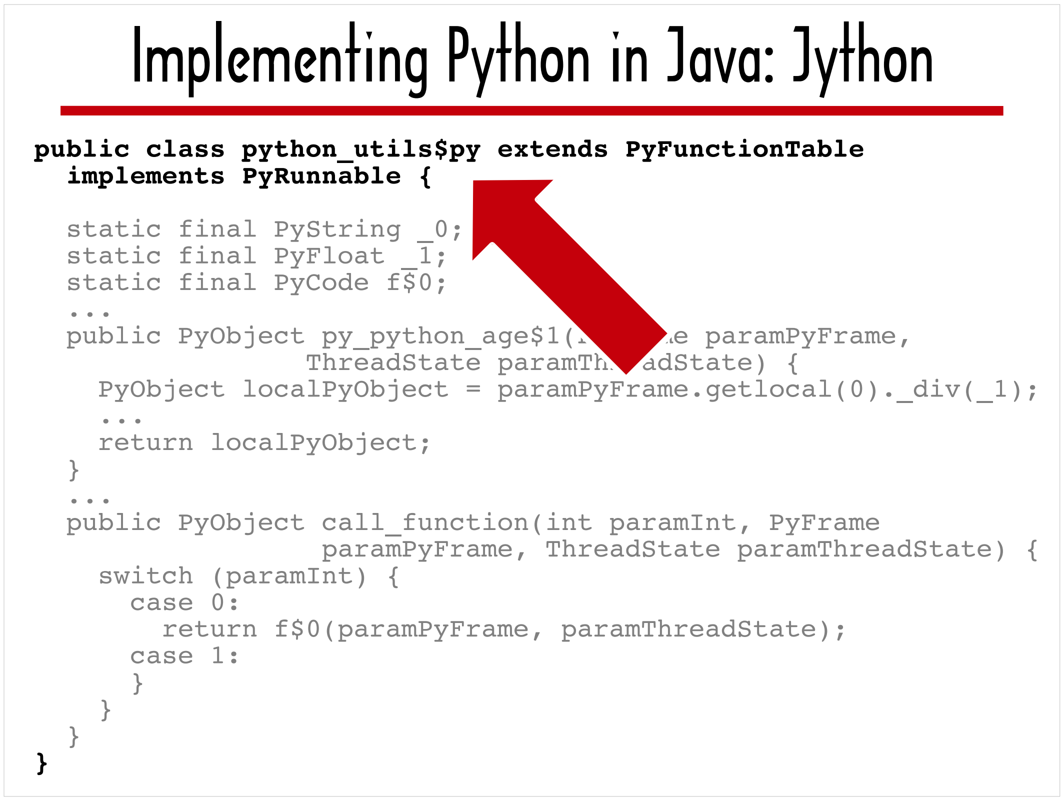
## Scala Hosting Jython: JSR 223

```
object PythonAge extends App {

  val scriptEngineManager = new ScriptEngineManage  )
  val pyEngine =
       scriptEngineManager.getEngineByName("pyt
  val engine =
       pyEngine.asInstanceOf[ScriptEngine wi      ocable]

  engine.eval(new FileReader("python_utils     ))

  val age =  9.0.asInstanceOf[AnyRef]
  val pythonAge = engine.invokeFunction("py_python_age",
                                       age)
  System.out.println(pythonAge.asInstanceOf[Double].round)

}
```

**`ScriptEngine/Invocable#invokeFunction`** is comparable to **`Jep#invoke`**: given the name of a function and a variable number of arguments typecast as Java objects, it executes the function.

Where **`JEPP#invoke`** uses the Python/C API behind the scenes, Jython has to jump through a few hoops in order to generate valid java bytecode for a function defined using Python syntax.

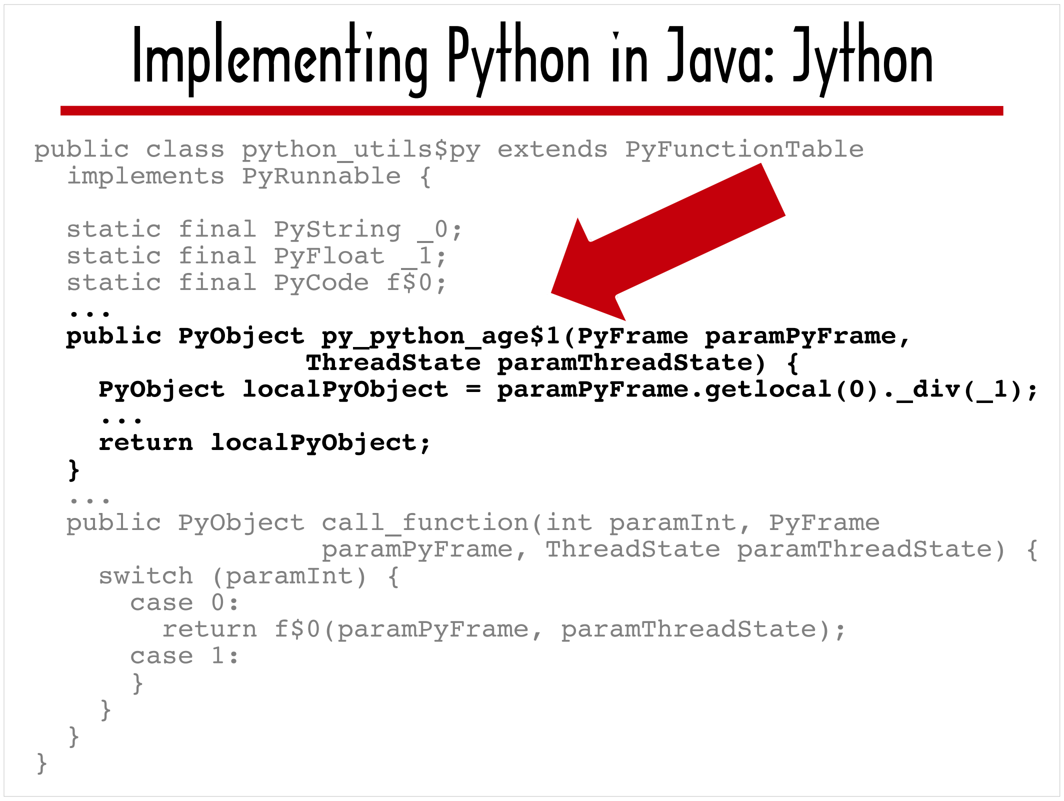
On the next couple of slides I'll show you some excerpts from a Java .class file generated by Jython so you can gain some insight into the Jython compilation process.



## Implementing Python in Java: Jython

```
public class python_utils$py extends PyFunctionTable
  implements PyRunnable {

  static final PyString _0;
  static final PyFloat _1;
  static final PyCode f$0;
  ...
  public PyObject py_python_age$1([illegible]e paramPyFrame,
                 ThreadState paramTh[illegible]adState) {
    PyObject localPyObject = paramPyFrame.getlocal(0)._div(_1);
    ...
    return localPyObject;
  }
  ...
  public PyObject call_function(int paramInt, PyFrame
                paramPyFrame, ThreadState paramThreadState) {
    switch (paramInt) {
      case 0:
        return f$0(paramPyFrame, paramThreadState);
      case 1:
      }
    }
  }
}
```

Jython coerces python_utils into being a Java class called python_utils$py ...



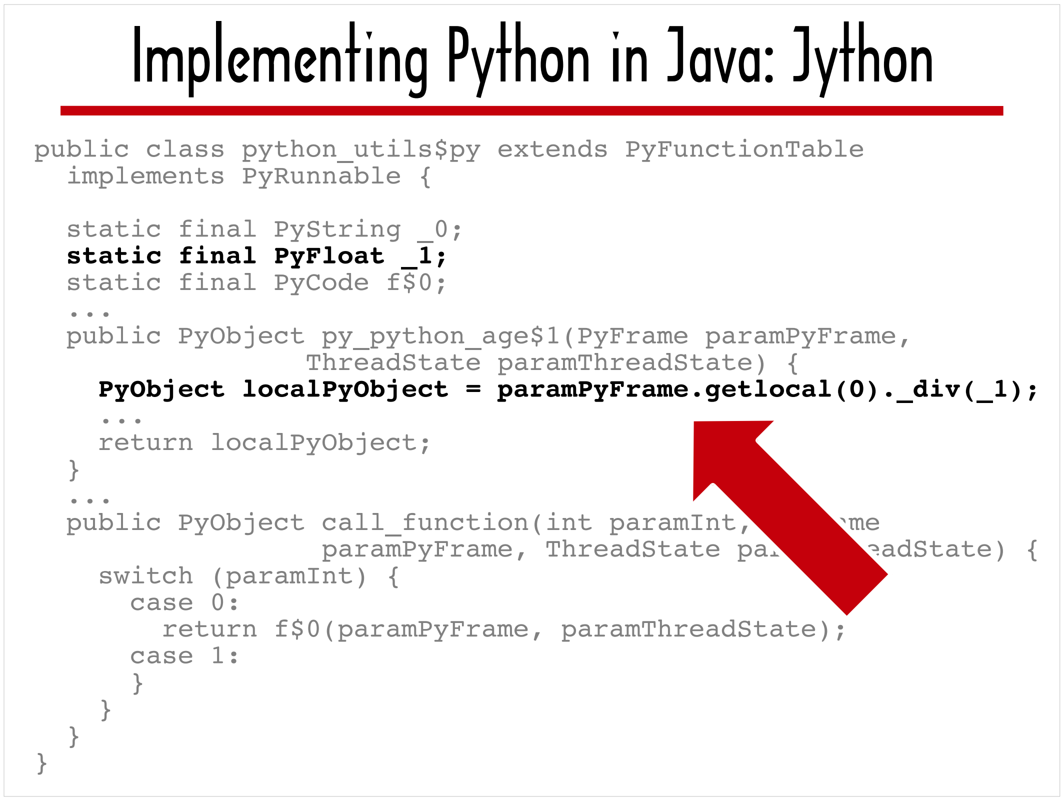
## Implementing Python in Java: Jython

```
public class python_utils$py extends PyFunctionTable
  implements PyRunnable {

  static final PyString _0;
  static final PyFloat _1;
  static final PyCode f$0;
  ...
  public PyObject py_python_age$1(PyFrame paramPyFrame,
                 ThreadState paramThreadState) {
    PyObject localPyObject = paramPyFrame.getlocal(0)._div(_1);
    ...
    return localPyObject;
  }
  ...
  public PyObject call_function(int paramInt, PyFrame
                  paramPyFrame, ThreadState paramThreadState) {
    switch (paramInt) {
      case 0:
        return f$0(paramPyFrame, paramThreadState);
      case 1:
      }
    }
  }
}
```

… with an instance method called **`py_python_age$1`** that returns a PyObject and takes a PyFrame argument and a ThreadState argument.

Until relatively recently, running on the JVM meant that the return type and the argument types needed to be specified at compile time. Jython deals with this by generating the same signature (a PyObject return value, a PyFrame argument and a ThreadState argument) for every method that represents a function defined using Python syntax, like **`py_python_age`**.

Note that I used the current stable version of Jython, which, at this writing was 3.5.3b1. Future versions of Jython will take advantage of changes to the JVM that make it a better platform for dynamic languages, which will improve performance and simplify the codebase.

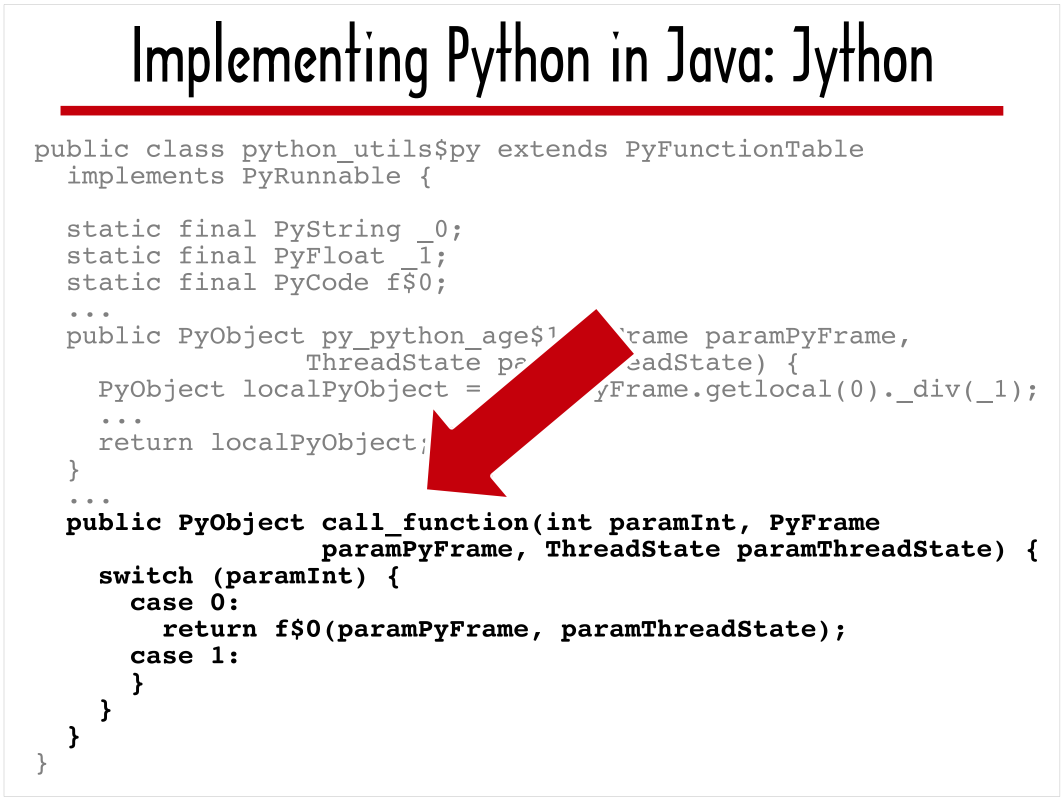


## Implementing Python in Java: Jython

```
public class python_utils$py extends PyFunctionTable
  implements PyRunnable {

  static final PyString _0;
  static final PyFloat _1;
  static final PyCode f$0;
  ...
  public PyObject py_python_age$1(PyFrame paramPyFrame,
                ThreadState paramThreadState) {
    PyObject localPyObject = paramPyFrame.getlocal(0)._div(_1);
    ...
    return localPyObject;
  }
  ...
  public PyObject call_function(int paramInt,       me
                 paramPyFrame, ThreadState pa      eadState) {
    switch (paramInt) {
      case 0:
        return f$0(paramPyFrame, paramThreadState);
      case 1:
      }
    }
  }
}
```

Here I have highlighted the Jython implementation of the “business logic” in **`py_python_age`**, dividing the passed in age by 2.8 (typecast as a PyFloat and represented by **`_1`**).

A PyFrame, such as **`paramPyFrame`**, stores information about and provides access to global and local variables.

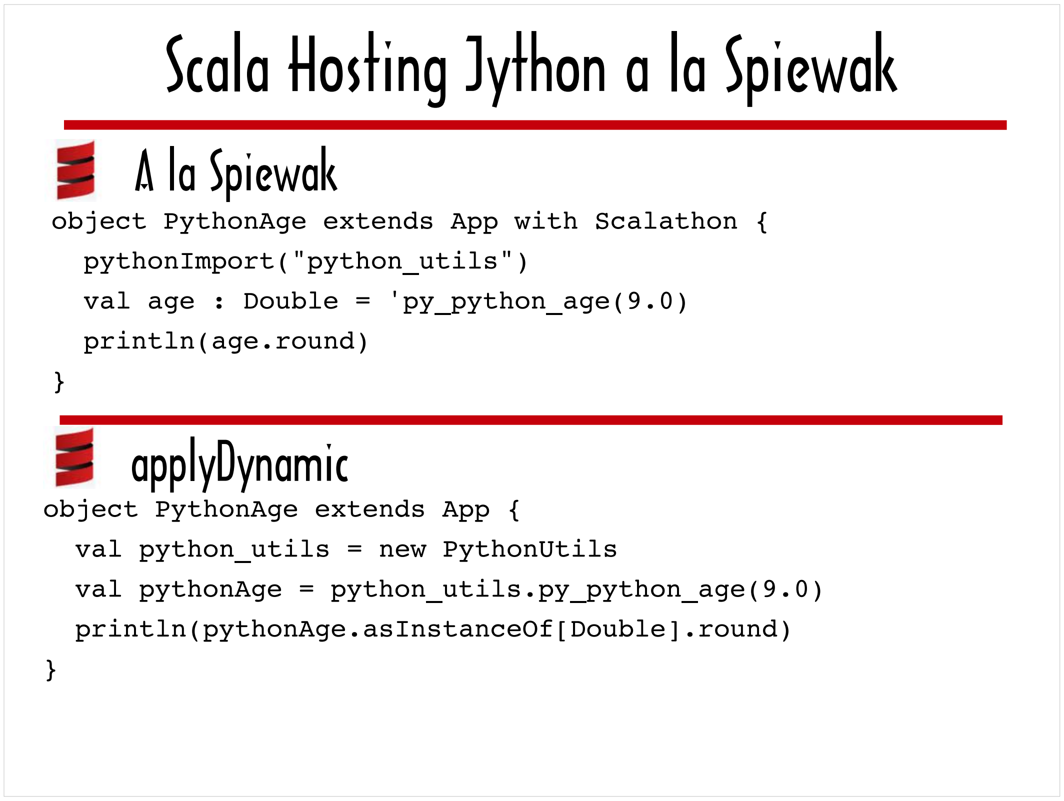


## Implementing Python in Java: Jython

```
public class python_utils$py extends PyFunctionTable
  implements PyRunnable {

  static final PyString _0;
  static final PyFloat _1;
  static final PyCode f$0;
  ...
  public PyObject py_python_age$1      rame paramPyFrame,
                 ThreadState pa      eadState) {
    PyObject localPyObject =      yFrame.getlocal(0)._div(_1);
    ...
    return localPyObject;
  }
  ...
  public PyObject call_function(int paramInt, PyFrame
                 paramPyFrame, ThreadState paramThreadState) {
    switch (paramInt) {
      case 0:
        return f$0(paramPyFrame, paramThreadState);
      case 1:
      }
    }
  }
}
```

For each function defined in a Python-based module, Jython generates a function called **`call_function`** that returns a PyObject and takes an int argument, a PyFrame argument and a ThreadState argument. The int argument represents a Jython-generated and assigned function ID#, and **`call_function`** switches on this function ID#.

When a function defined in a Python module is invoked, Jython invokes **`call_function`** on the instance of the class it generates to represent the module (eg python_utils$py), which passes control to the specified function.



## Scala Hosting Jython a la Spiewak

### A la Spiewak

```
object PythonAge extends App with Scalathon {
  pythonImport("python_utils")
  val age : Double = 'py_python_age(9.0)
  println(age.round)
}
```

### applyDynamic

```
object PythonAge extends App {
  val python_utils = new PythonUtils
  val pythonAge = python_utils.py_python_age(9.0)
  println(pythonAge.asInstanceOf[Double].round)
}
```

The Jython 'Out of the Box' code can be wrapped in a Spiewak-inspired way (as shown of the top of this slide) or using an applyDynamic-based wrapper (as shown on the bottom).

In each case the implementation is very similar to the JEPP versions of these wrappers I walked through, so I am not including the source for these wrappers in this slide deck.

However, I did post the source for these Jython wrappers along with the rest of the source covered in this talk at https://github.com/A-OK/Snakes-and-Ladders

## Python hosting Scala

In this next section, we'll look at examples of Python/Scala interop where Python is the host language and Scala is the guest.

Just as I accessed functions from a  Python micro-library in the “Scala hosting Python” section, I'm going to using a Scala micro-library to represent a useful library written in Scala.

And just as I used **`py_python_age`** to represent useful Python logic, I am going to use a similarly pneumonic and symbolic function written in Scala.

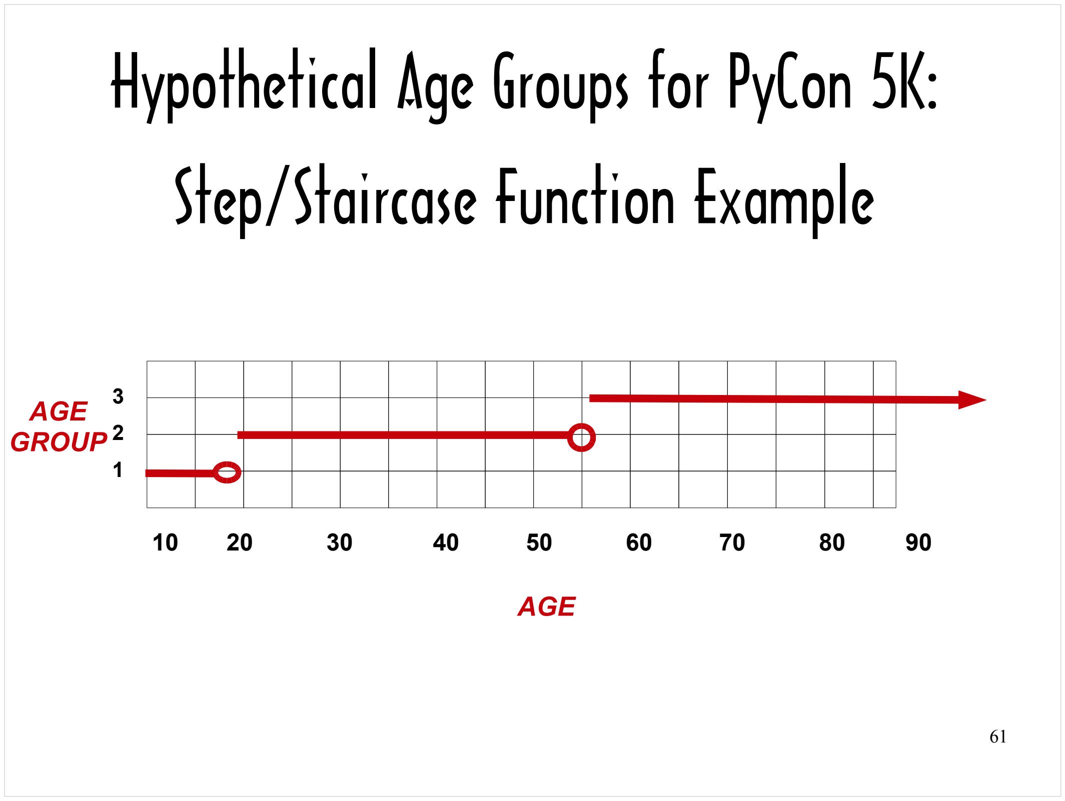
PyCon 2012 featured the first ever PyCon 5K Run.

The runners were not broken down by age group, but if they had been a function like the core function in my Scala micro-library could have been used to determine a particular participant's age group.

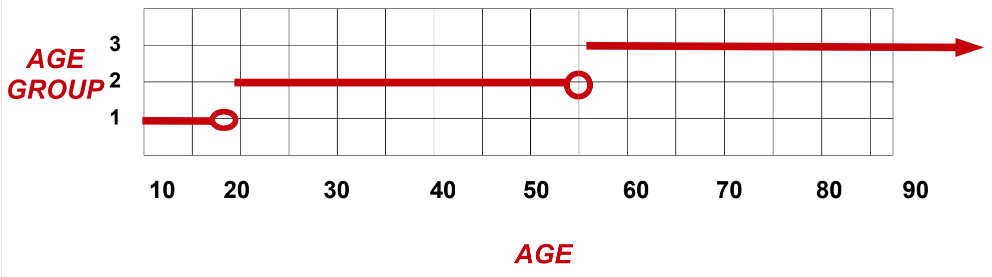
The function I wrote to represent useful Scala logic takes an age as a parameter and returns one of three age groups: group 1 for those 19 and under, group 2 for those between 20 and 55, and group 3 for those over 55.

What makes this a symbolic choice for a Scala micro-library?

It's an example of a step function, also known as a staircase function, and “scala” means staircase in Italian.



## Hypothetical Age Groups for PyCon 5K: Step/Staircase Function Example



61

Because staircase function returns the same value for an interval of argument values and jumps to the next interval for the next set of argument values, its graph resembles a staircase.

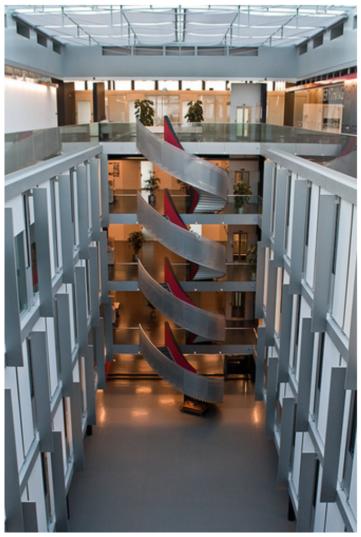
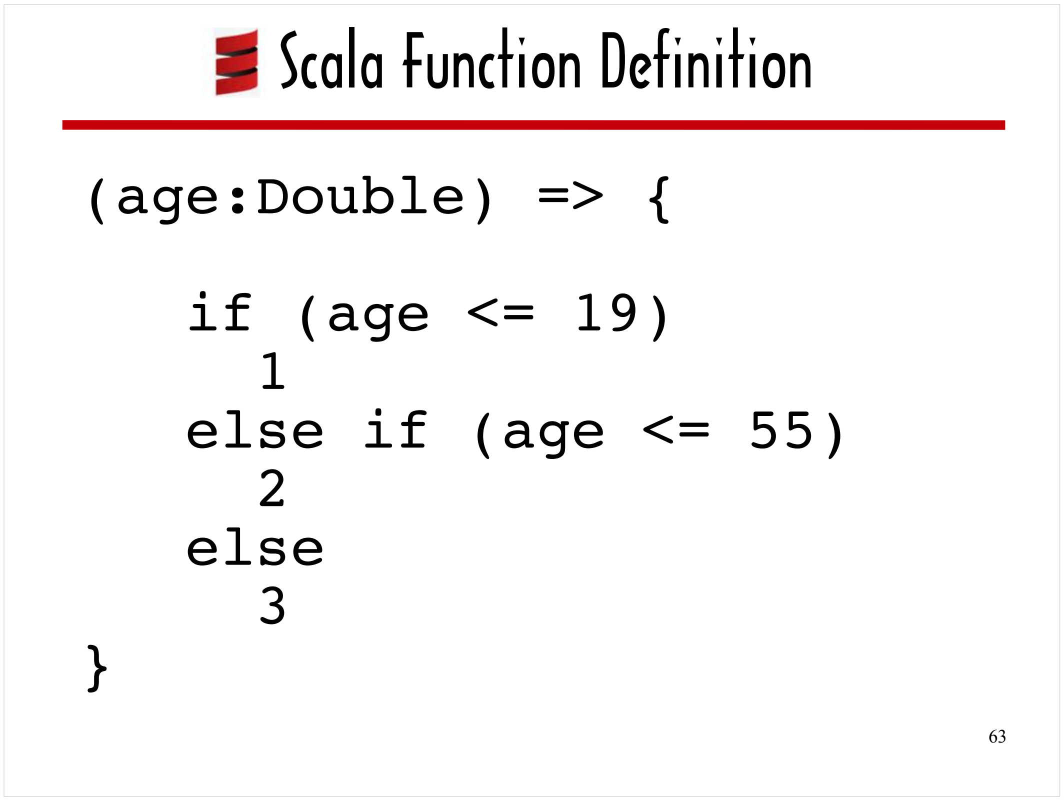


Photo by Miles Sabin: http://www.flickr.com/photos/montpelier/3957416434/

The Scala logo is based on a staircase at the École Polytechnique Fédérale de Lausanne, where Scala was created.

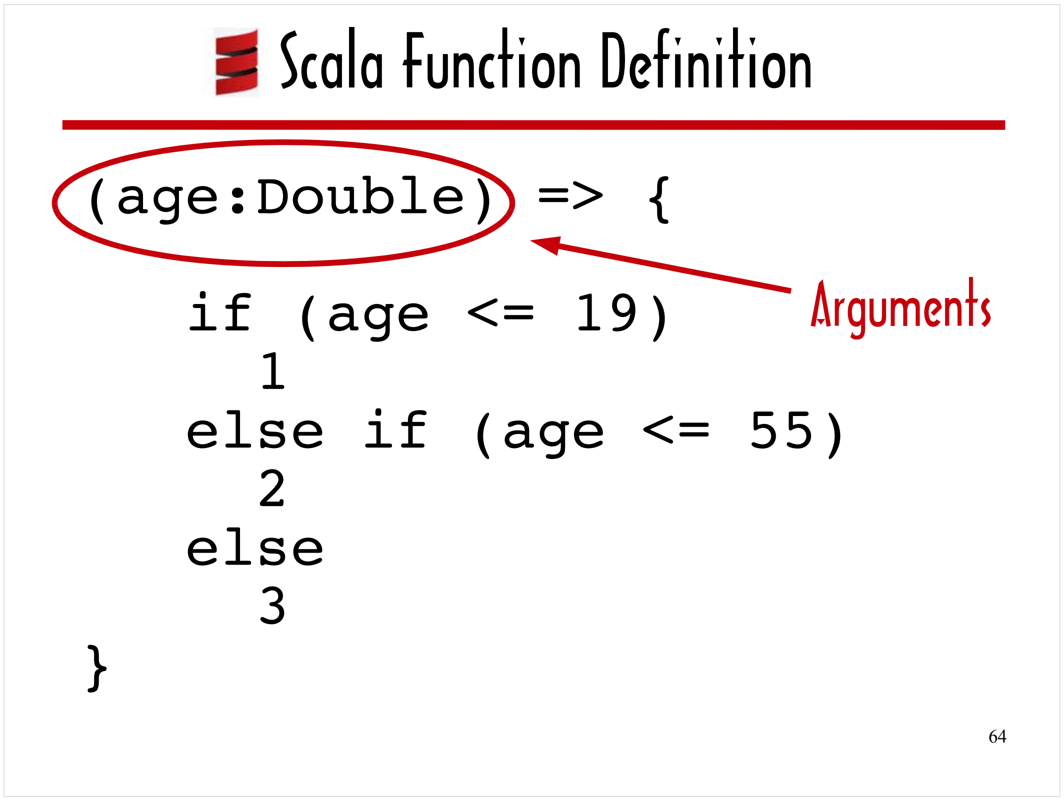


## Scala Function Definition

```
(age:Double) => {

   if (age <= 19)
     1
   else if (age <= 55)
     2
   else
     3
}
```

63

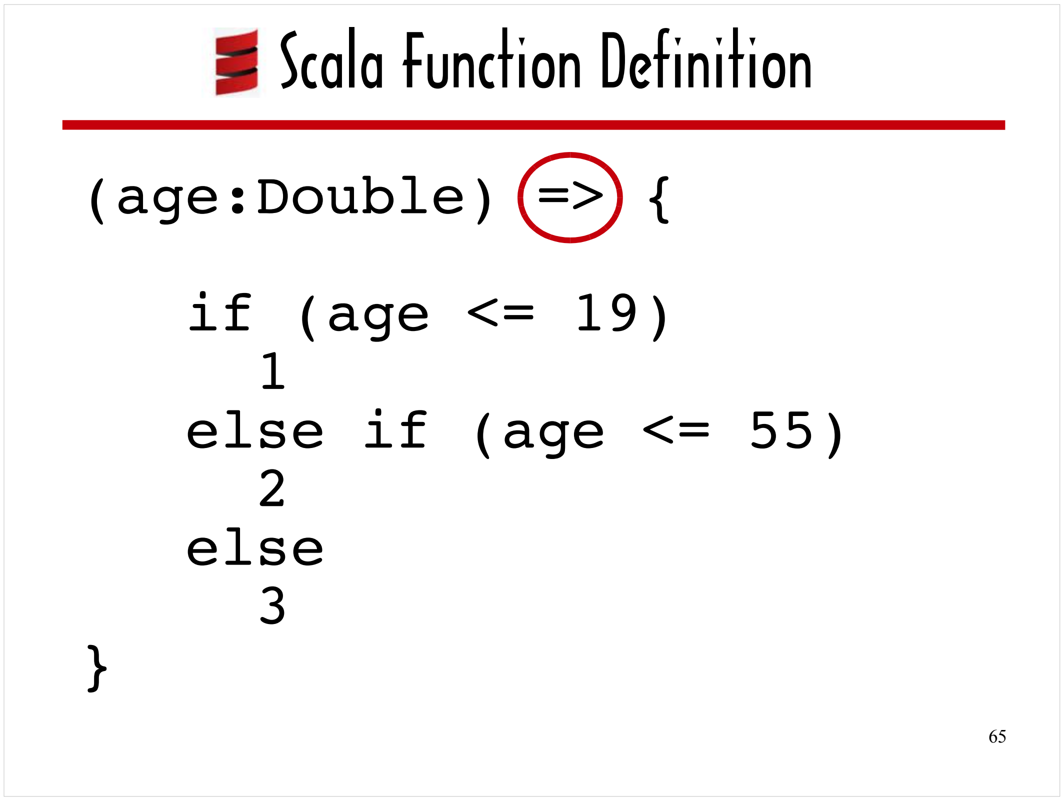
Here's my age group function “step” function using  Scala anonymous function syntax:



## Scala Function Definition

```
(age:Double) => {
    if (age <= 19)
      1
    else if (age <= 55)
      2
    else
      3
}
```

The argument list is specified between parenthesis, and the name of each argument is followed by a colon and the argument type.

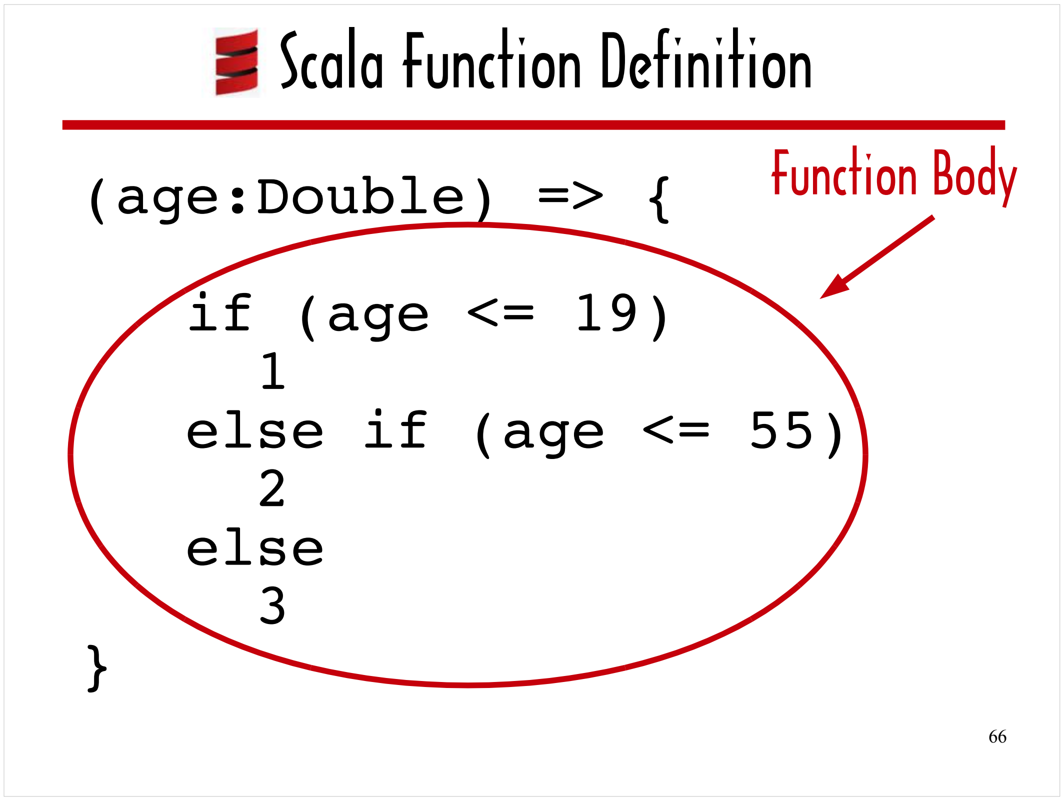


## Scala Function Definition

```
(age:Double) => {

   if (age <= 19)
     1
   else if (age <= 55)
     2
   else
     3
}
```

65

A arrow separates the function body from argument list.

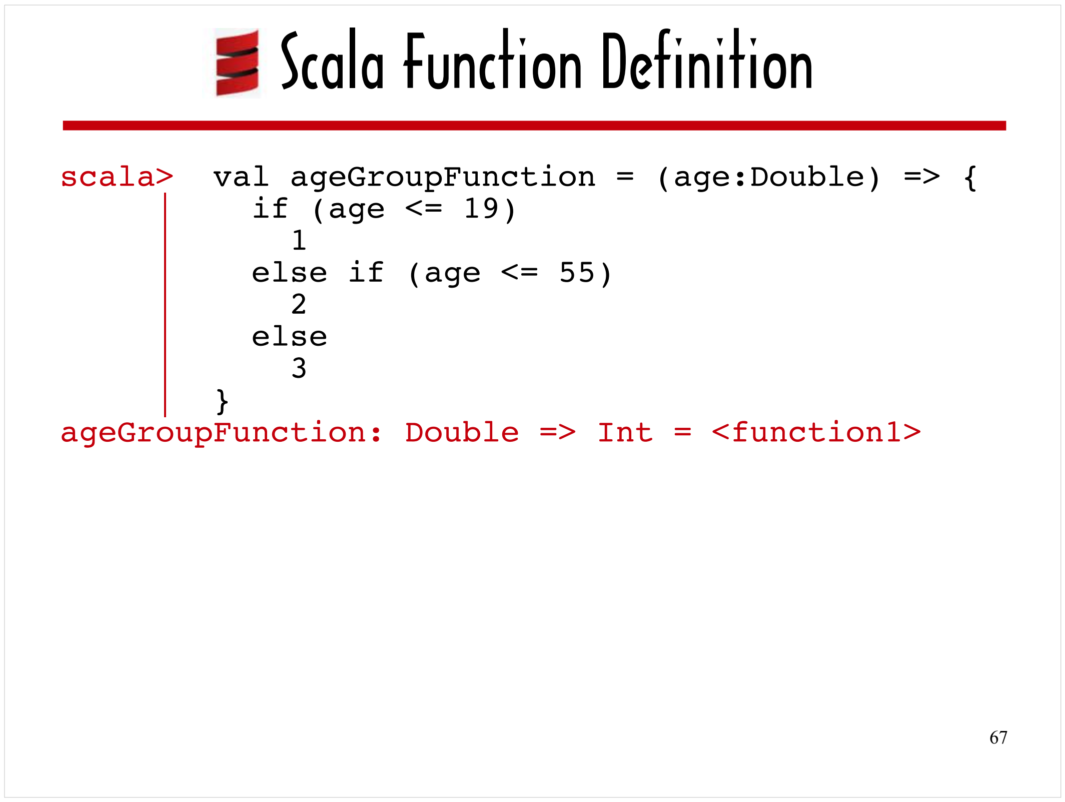


## Scala Function Definition

```
(age:Double) => {
    if (age <= 19)
      1
    else if (age <= 55)
      2
    else
      3
}
```

Function Body

The function body for this function is flanked by curly braces because it spans multiple lines. The curly braces are not required for one-line functions.

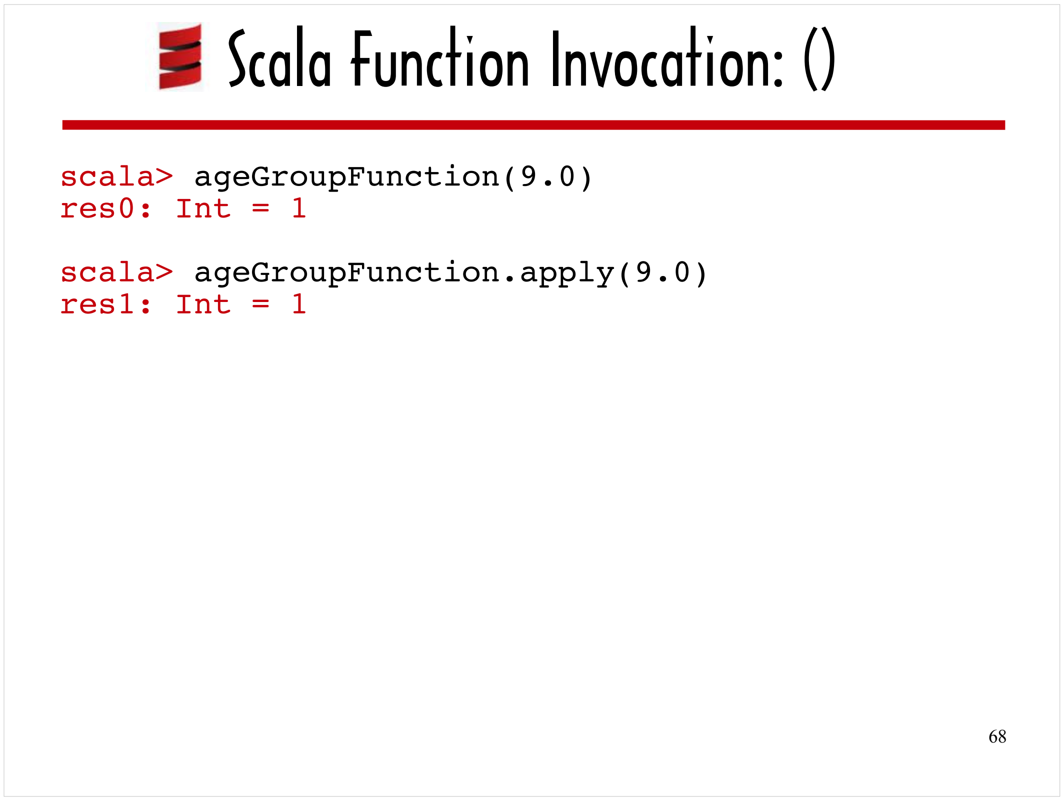


## Scala Function Definition

```
scala>  val ageGroupFunction = (age:Double) => {
          if (age <= 19)
            1
          else if (age <= 55)
            2
          else
            3
        }
ageGroupFunction: Double => Int = <function1>
```

Here I've assigned the function to the variable **`ageGroupFunction`** in the Scala interactive console.

As a explained when we first looked at the Scala console, the Scala console reports the type of each entered expression. The notation **`Double => Int`** in the console output indicates that the entered function takes a Double as its only argument and returns an Int. Specifying a return type for a function (by following the argument list with a colon and the return type) is optional.
Scala infers the return type when it is not specified.

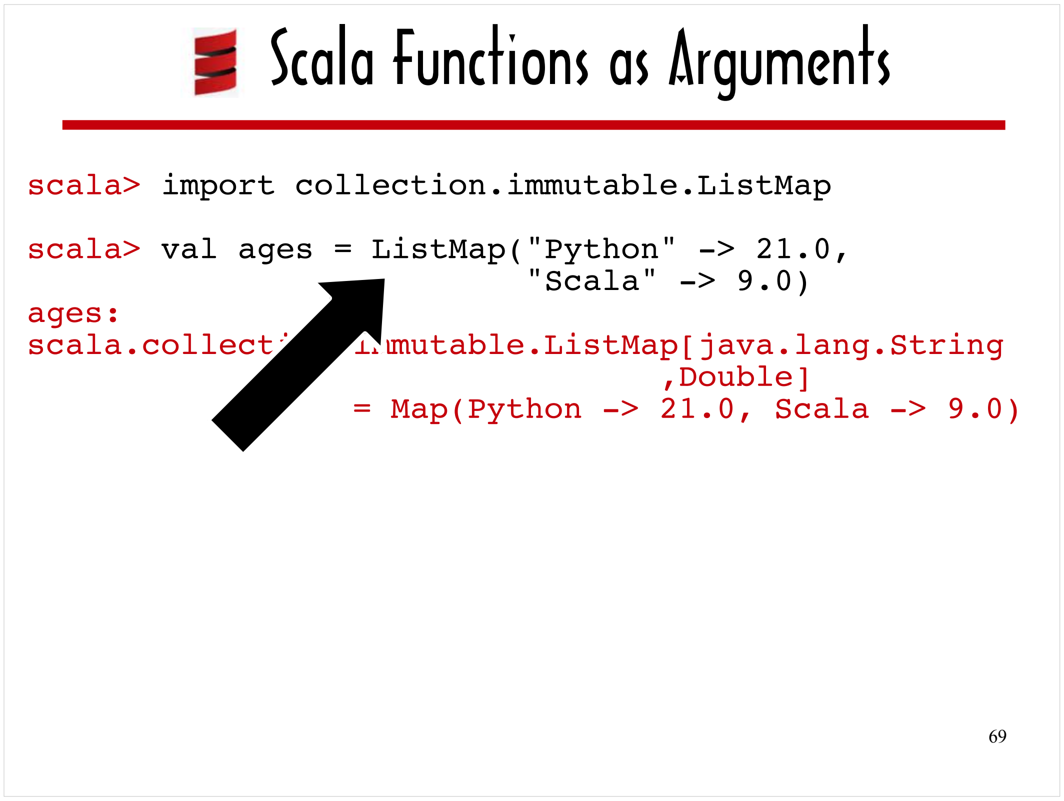


## Scala Function Invocation: ()

```
scala> ageGroupFunction(9.0)
res0: Int = 1

scala> ageGroupFunction.apply(9.0)
res1: Int = 1
```

As I've noted earlier, function invocation in Scala is similar to function invocation in Python. You can follow the function name with any arguments flanked by parentheses or you can use the **`apply`** method. Scala's **`apply`** is comparable to Python's **`__call__`**.

You can see that if Scala (which is a 9-years old) had run in the PyCon 5K and The participants had been placed into the hypothetical age groups I set up, Scala would have been in Age Group #1.



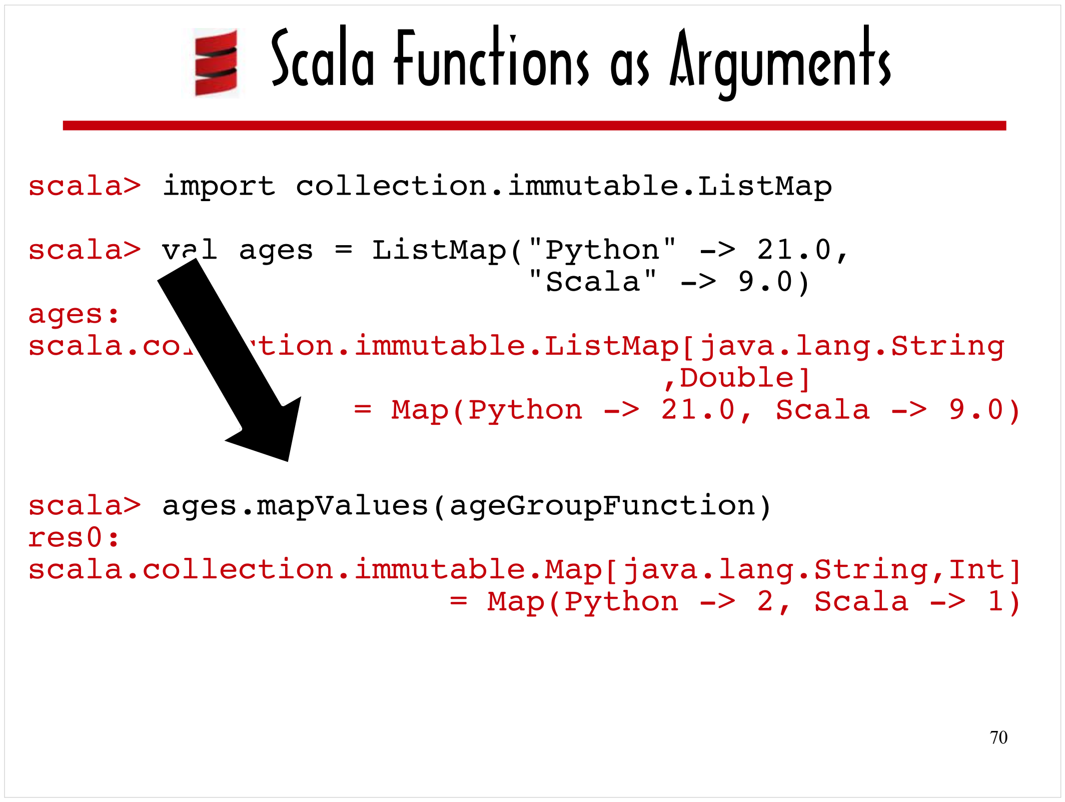
## Scala Functions as Arguments

```
scala> import collection.immutable.ListMap

scala> val ages = ListMap("Python" -> 21.0,
                          "Scala" -> 9.0)
ages:
scala.collecti     immutable.ListMap[java.lang.String
                                 ,Double]
                 = Map(Python -> 21.0, Scala -> 9.0)
```

In Scala, as in Python, a function can be passed to another function as an argument.

In the simulated console in the next slide, I'll pass **`ageGroupFunction`** to **`ListMap#mapValues`**. A ListMap is a data structure that holds key/value pairs. **`ListMap#mapValues`** is comparable to **`map`** in Python: it returns a new collection comprised of the results of passing each item in the collection to the supplied function.

Before I can call **`ListMap#mapValues`**, I need an instance of ListMap. I created one that that pairs “Python” and “Scala” with their ages, 21 and 9, respectively – and I assigned it to **`ages`**.

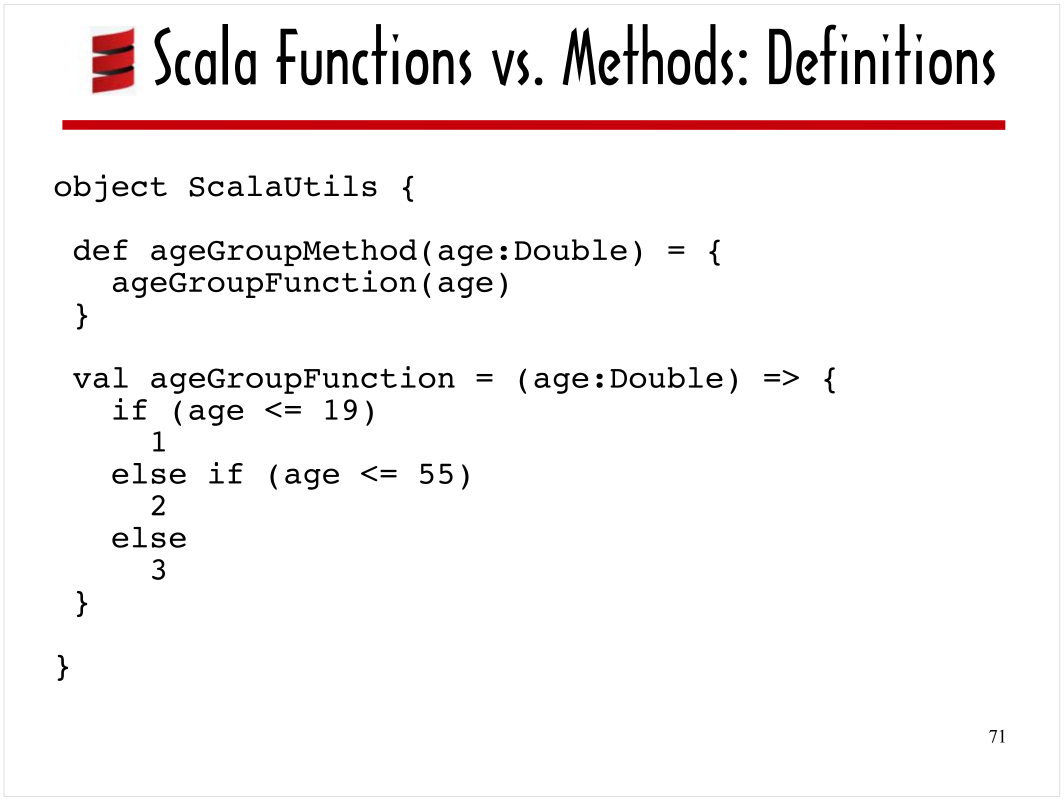


## Scala Functions as Arguments

```
scala> import collection.immutable.ListMap

scala> val ages = ListMap("Python" -> 21.0,
                          "Scala" -> 9.0)
ages:
scala.collection.immutable.ListMap[java.lang.String
                                  ,Double]
                 = Map(Python -> 21.0, Scala -> 9.0)


scala> ages.mapValues(ageGroupFunction)
res0:
scala.collection.immutable.Map[java.lang.String,Int]
                      = Map(Python -> 2, Scala -> 1)
```

As you can see, calling **`mapValues`** on **`ages`** yields a ListMap where the keys are the language names and the values are the respective age groups they'd land in had they been runners in the PyCon 5K.

Shortly, we'll look at how to access a ListMap defined in Scala from the Jython interactive console, and I'll show you how to pass a function defined using Python syntax to **`ListMap#mapValues.`**



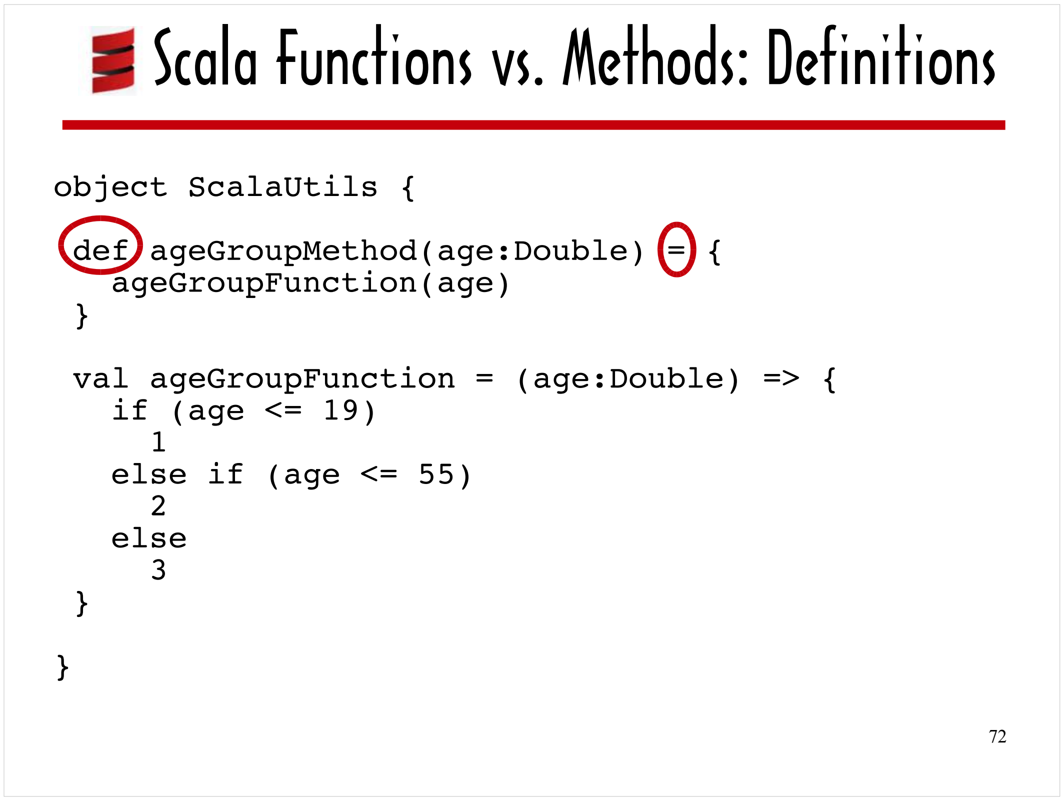
## Scala Functions vs. Methods: Definitions

```
object ScalaUtils {

 def ageGroupMethod(age:Double) = {
   ageGroupFunction(age)
 }

 val ageGroupFunction = (age:Double) => {
   if (age <= 19)
     1
   else if (age <= 55)
     2
   else
     3
 }

}
```

Before showing you how to access Scala code from Python, I would like to go over a few aspects of Scala method syntax, particularly where it differs from Scala function syntax.

I think it's important to be aware of some of the differences between Scala methods and functions, so I decided to include both kinds of constructs in my Scala micro-library. If you're going to be hosting Scala from Python, you're probably going to need to know how to work with Scala methods as well as Scala functions.

The ScalaUtils singleton object includes the **`ageGroupFunction`** we've been looking at, and an **`ageGroupMethod`** instance method that does nothing other than wrap **`ageGroupFunction`**. The main purpose of **`ageGroupMethod`** is to help me demonstrate method syntax vs. function syntax.



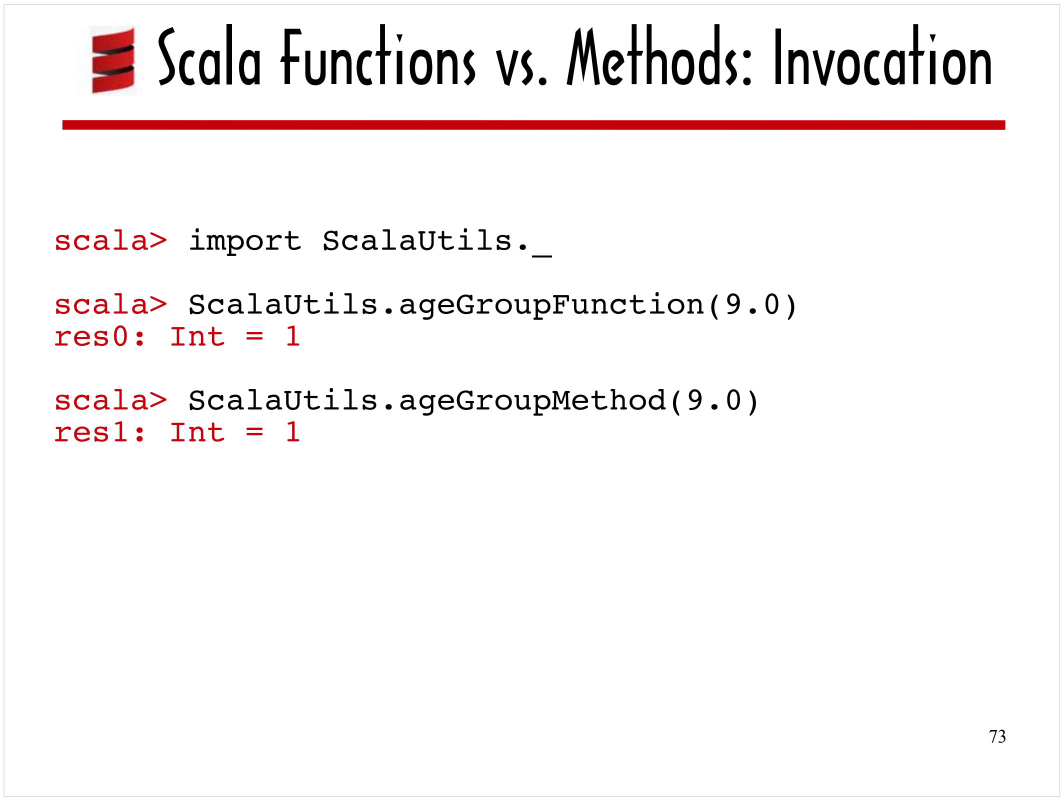
## Scala Functions vs. Methods: Definitions

```
object ScalaUtils {

 def ageGroupMethod(age:Double) = {
   ageGroupFunction(age)
 }

 val ageGroupFunction = (age:Double) => {
   if (age <= 19)
     1
   else if (age <= 55)
     2
   else
     3
 }

}
```

72

A method definition begins with the keyword **`def`**.

An equals sign separates the method signature from the method body.

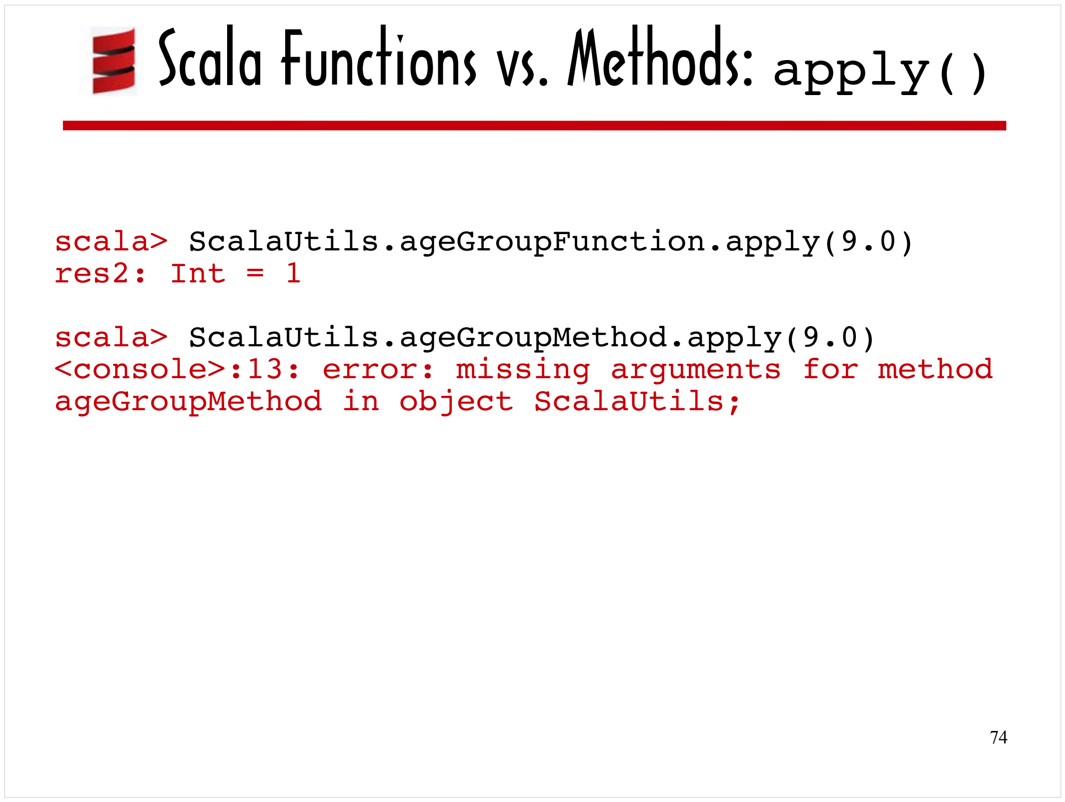


## Scala Functions vs. Methods: Invocation

```
scala> import ScalaUtils._

scala> ScalaUtils.ageGroupFunction(9.0)
res0: Int = 1

scala> ScalaUtils.ageGroupMethod(9.0)
res1: Int = 1
```

As noted earlier, you can call methods on or access variables on a singleton object without instantiating it.

As you can see from this simulated Scala console session, you can invoke **`ageGroupFunction`** and **`ageGroupMethod`** using standard dot notation.

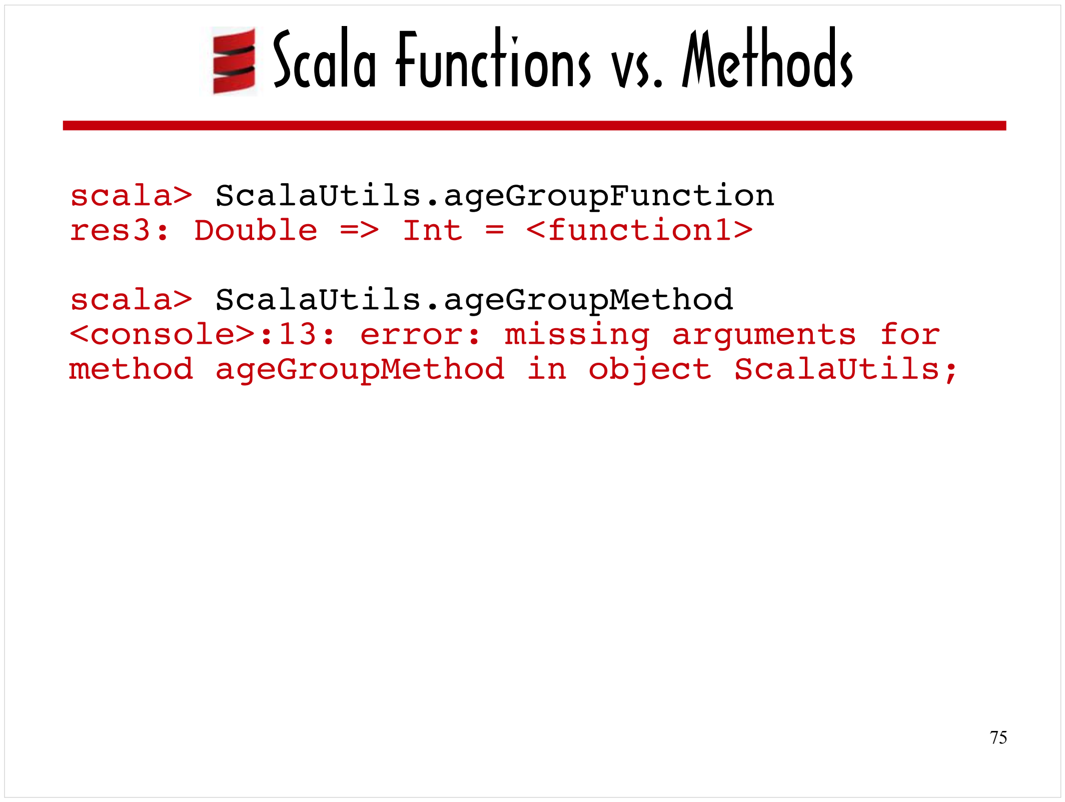


## Scala Functions vs. Methods: `apply()`

```
scala> ScalaUtils.ageGroupFunction.apply(9.0)
res2: Int = 1

scala> ScalaUtils.ageGroupMethod.apply(9.0)
<console>:13: error: missing arguments for method
ageGroupMethod in object ScalaUtils;
```

We've seen that the logic in the body of a Scala function is executed when **`apply`** is invoked on the function, and I've mentioned that apply is comparable to Python's **`__call__`**.

As you can see here apply is not supported for methods. Trying to call **`apply`** on **`ageGroupMethod`** yields error output in the console.



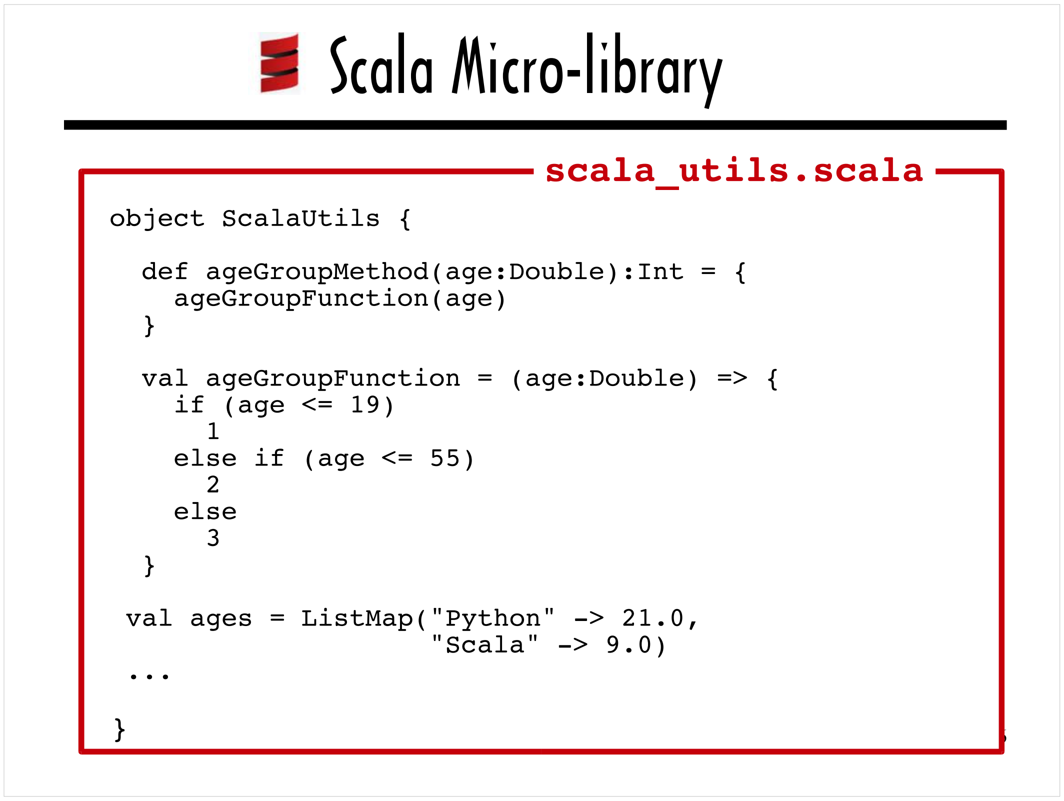
## Scala Functions vs. Methods

```
scala> ScalaUtils.ageGroupFunction
res3: Double => Int = <function1>

scala> ScalaUtils.ageGroupMethod
<console>:13: error: missing arguments for
method ageGroupMethod in object ScalaUtils;
```

A Scala function Is an object.

When you enter a function identifier in the console sans argument list, the console has no problem evaluating the entered expression. Scala recognizes that **`ScalaUtils.ageGroupFunction`** is a function that takes a Double and returns an Int.

As you can see here, a Scala method is only considered valid as part of a method invocation expression. When you try entering **`ScalaUtils.ageGroupMethod`** sans argument list, the console complains.



## Scala Micro-library

scala_utils.scala

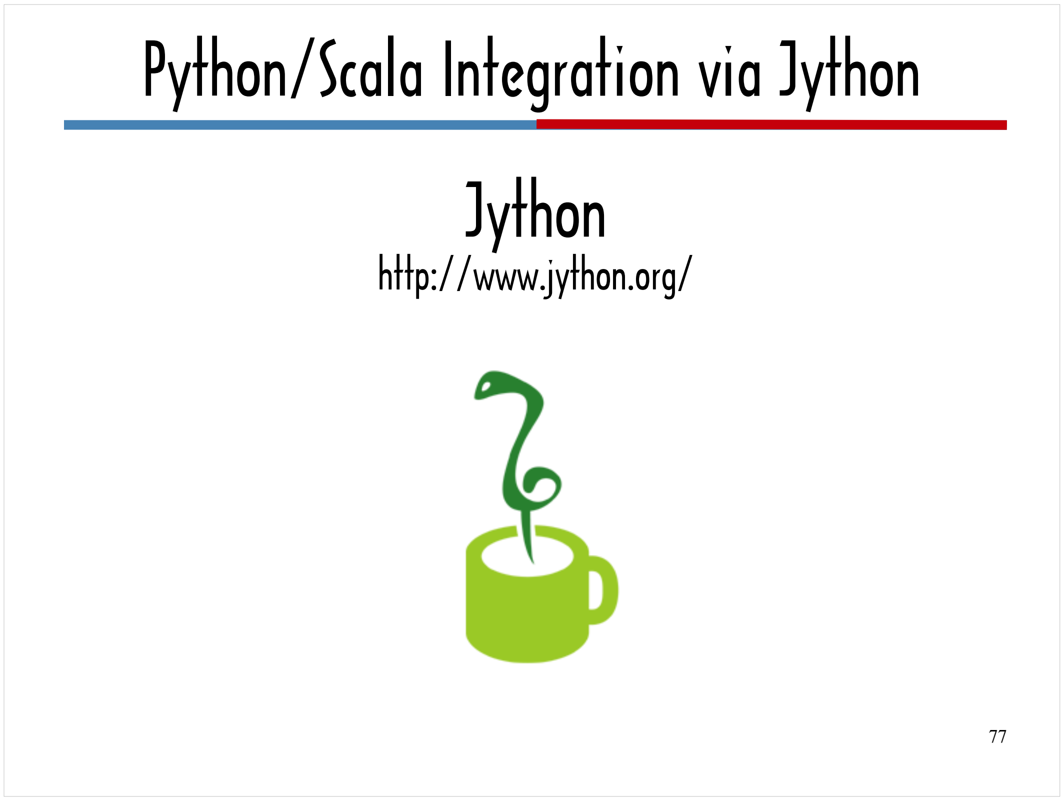
```
object ScalaUtils {

  def ageGroupMethod(age:Double):Int = {
    ageGroupFunction(age)
  }

  val ageGroupFunction = (age:Double) => {
    if (age <= 19)
      1
    else if (age <= 55)
      2
    else
      3
  }

 val ages = ListMap("Python" -> 21.0,
                    "Scala" -> 9.0)
 ...

}
```

Here is the Scala micro-library that represents useful Scala logic .

The ellipses following the **`ages`** ListMap indicates that there is additional code in the micro-library. I'll discuss the additional library code a bit later.

For now, I will focus on **`ageGroupMethod`**, **`ageGroupFunction`** and the **`ages`** ListMap.

I showed you to to use each of these in a Scala program. In the next group of slides, I'll show you how to access these ...



## Python/Scala Integration via Jython

Jython

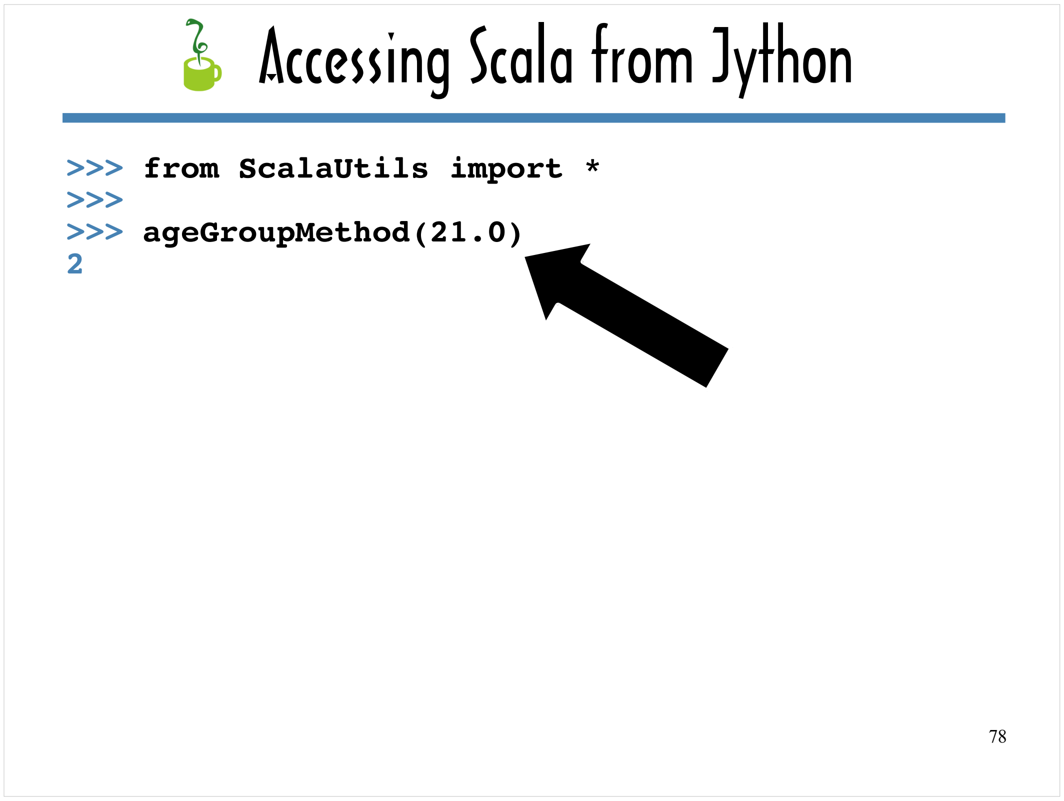
http://www.jython.org/

...from Jython.

In the Scala console, I showed you an example of passing a Scala function to another Scala function. I passed **`ageGroupFunction`** to **`ListMap#mapValues`**.

This next section is where I will show you how tight Scala/Jython integration can be. I'll show you how to “scalafy” a function defined using Python syntax so that it will be accepted as a valid argument to Scala functions that take Scala functions as arguments.

Following the section on Jython hosting Scala, I will cover several projects that make it possible for Python (ie the C-based implementation of Python) to host Scala.



## Accessing Scala from Jython

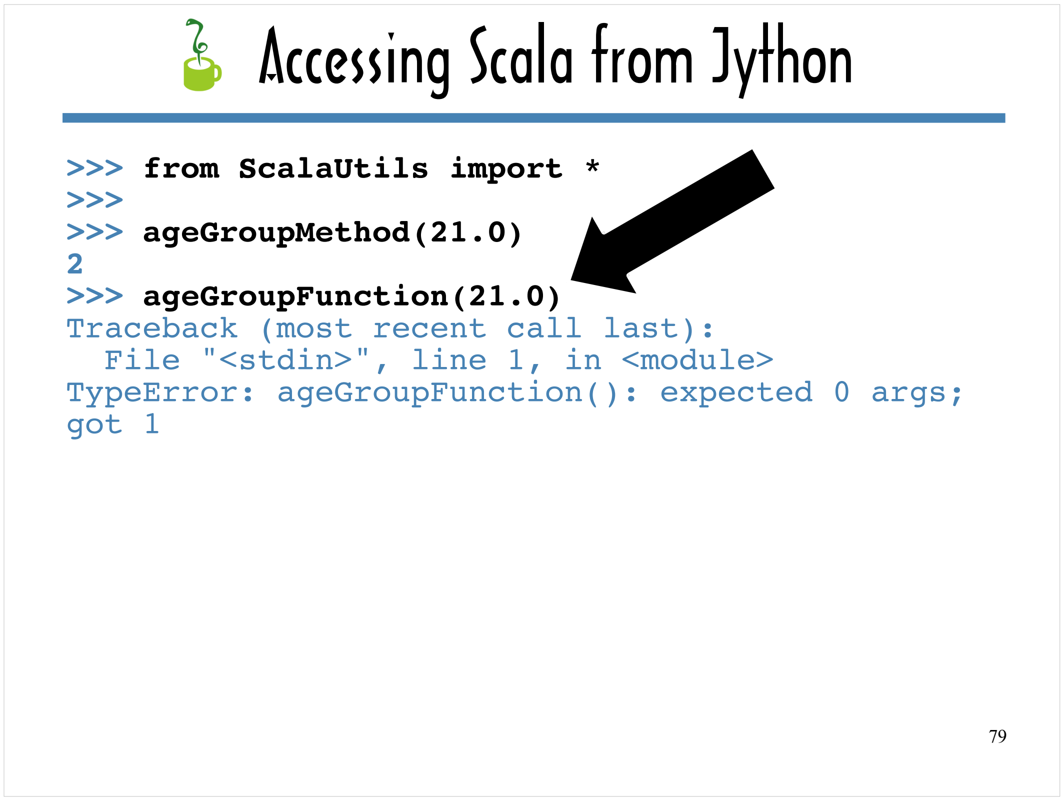
```
>>> from ScalaUtils import *
>>>
>>> ageGroupMethod(21.0)
2
```

78

Many of the slides in this section will simulate a Jython interactive console session. Entered text will be black. The prompt and any console output will be blue.

A Scala singleton object can be imported using the same syntax you would use to import a Python module, and as you can see, you can invoke **`AgeGroupMethod`** (which is written in Scala) as if it were a function defined using Python syntax.

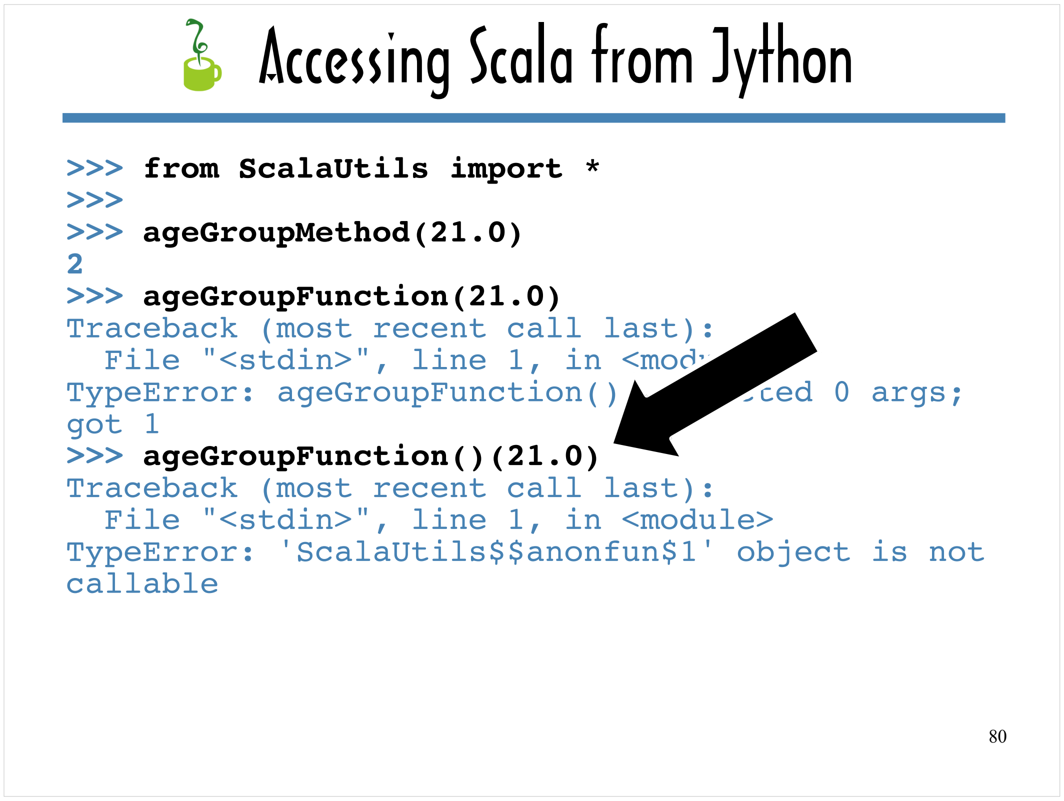
If you import everything defined in ScalaUtils with **`from ScalaUtils import *`**, you can then access Scala methods and instance variables in the Jython console  without prefixing them with “ScalaUtils”.



## Accessing Scala from Jython

```
>>> from ScalaUtils import *
>>>
>>> ageGroupMethod(21.0)
2
>>> ageGroupFunction(21.0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: ageGroupFunction(): expected 0 args;
got 1
```

If you try to call a Scala function as if it were defined using Python syntax, however, the console complains.



## Accessing Scala from Jython

```
>>> from ScalaUtils import *
>>>
>>> ageGroupMethod(21.0)
2
>>> ageGroupFunction(21.0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <mod
TypeError: ageGroupFunction()       cted 0 args;
got 1
>>> ageGroupFunction()(21.0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'ScalaUtils$$anonfun$1' object is not
callable
```

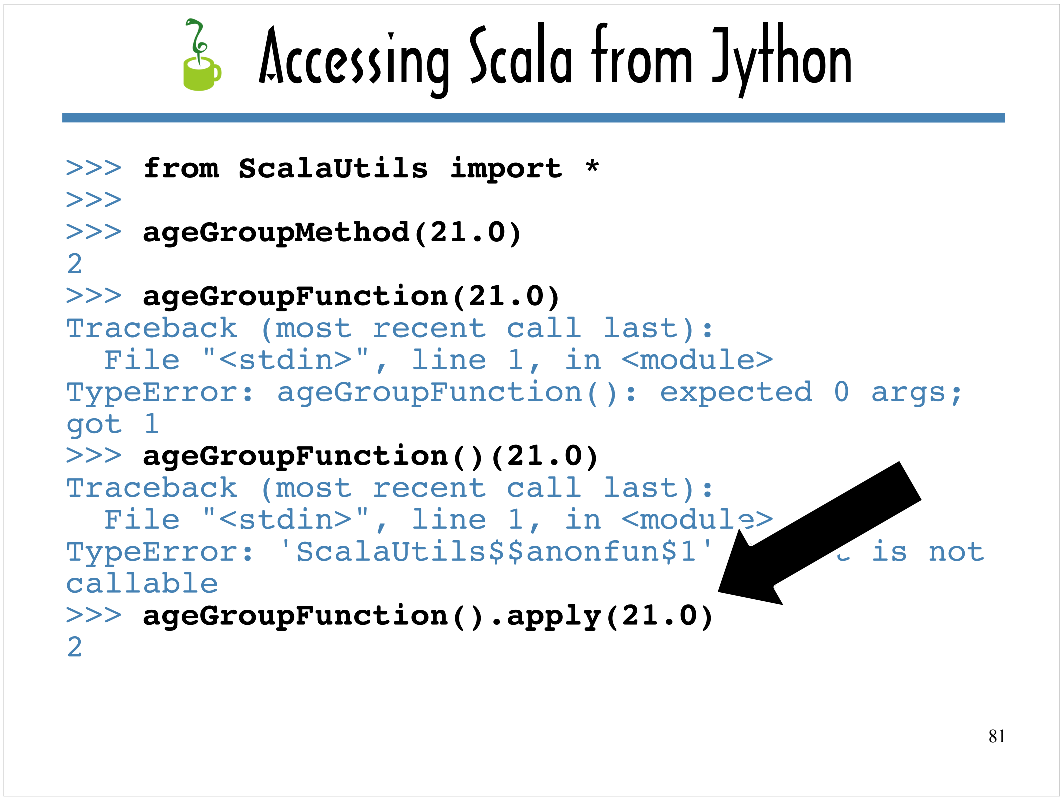
Before you can can invoke a Scala function from Jython, you need to suffix it with parentheses – as if it were a class that needed to be instantiated.

The reason for this has to do with the fact that Jython is implemented in Java, not Scala. To a large extent techniques for integrating Java and Jython work with Scala out of the box. Sometimes some hacking is required to Scala and Jython to work together.

When you compile Scala source code that includes a function, the Scala compiler creates a class with an **`apply`** method to represent the function. The function body serves as the method body for generated **`apply`** method. Recall that Scala's **`apply`** is comparable to Python's **`__call__`**, and that when you follow a function name with an argument list in Scala, Scala invokes the **apply** method on that function.

Jython recognizes that the class the Scala compiler generates represents a function, but it does not automatically create an instance of the function the way the Scala runtime would.

As you can see here, though, just following the function name with parentheses and following the “instantiated” function object with an argument list, yields console errors. Something else is required before you can use the function.

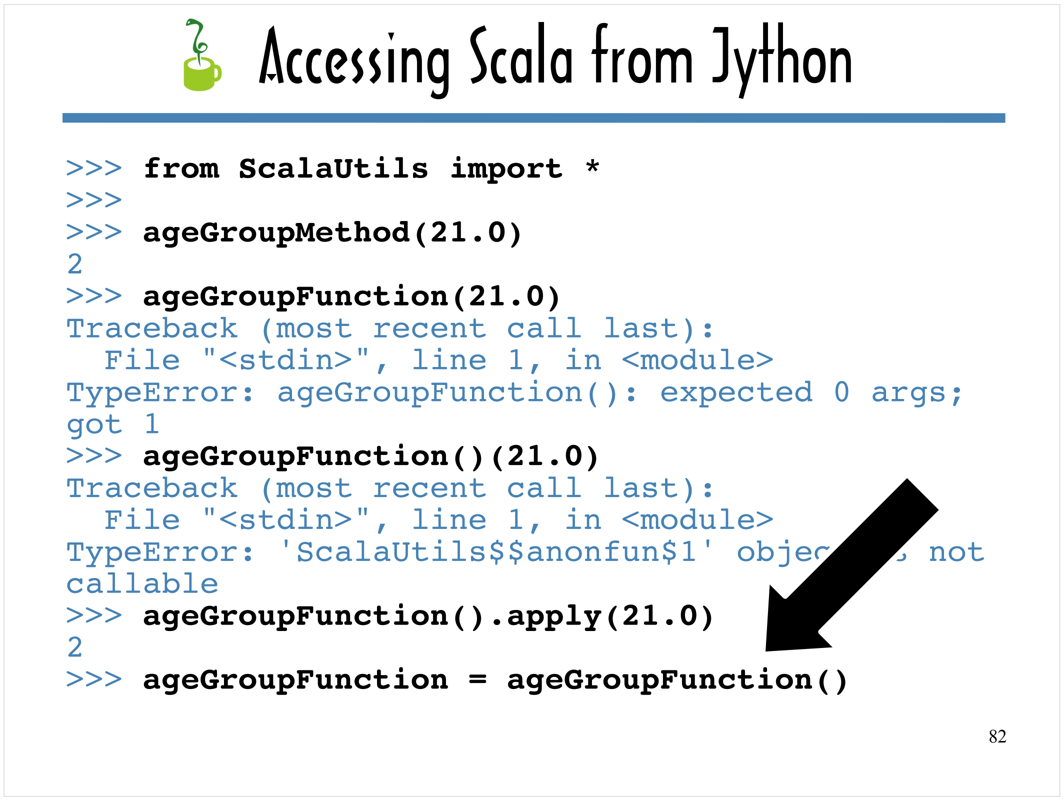


## Accessing Scala from Jython

```
>>> from ScalaUtils import *
>>>
>>> ageGroupMethod(21.0)
2
>>> ageGroupFunction(21.0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: ageGroupFunction(): expected 0 args;
got 1
>>> ageGroupFunction()(21.0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'ScalaUtils$$anonfun$1' [illegible]t is not
callable
>>> ageGroupFunction().apply(21.0)
2
```

To successfully invoke a function defined in Scala from Jython, you need to append parentheses to the function identifier, and then call **`apply`**, passing **`apply`** the argument list.

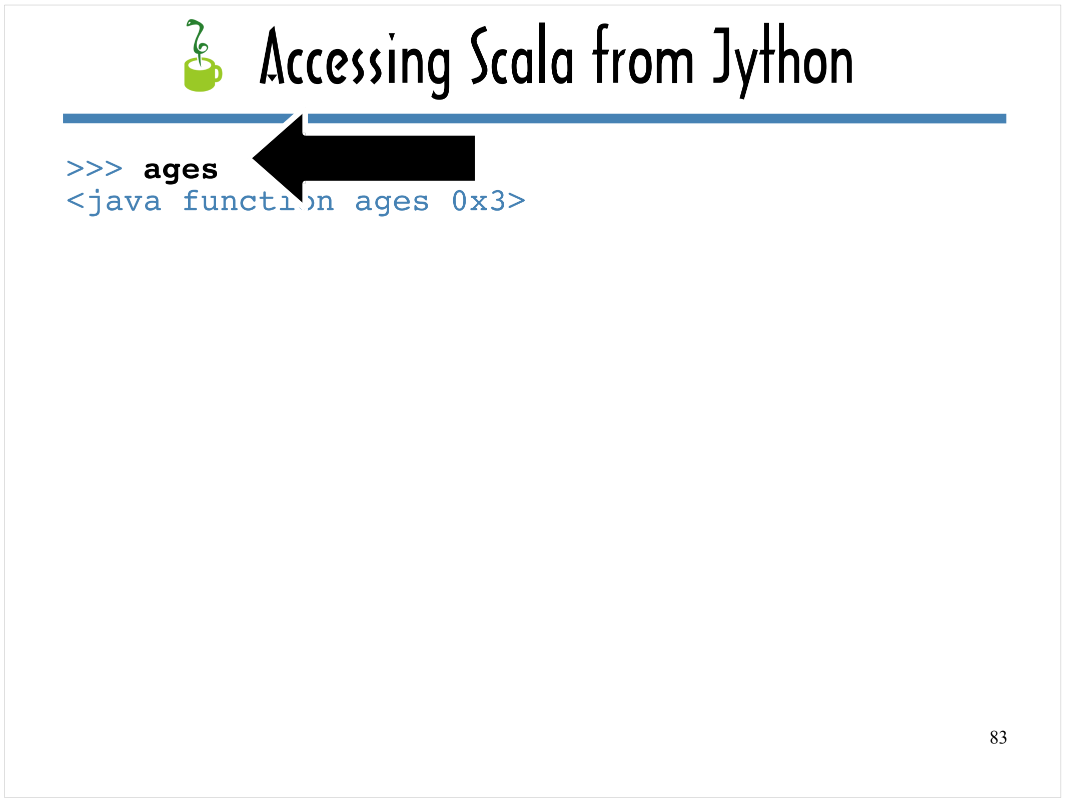
Scala will automatically call a function's **`apply`** method when it sees an argument list adjacent to a function identifier, but since Jython does not know it needs to do this, the developer needs to.



## Accessing Scala from Jython

```
>>> from ScalaUtils import *
>>>
>>> ageGroupMethod(21.0)
2
>>> ageGroupFunction(21.0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: ageGroupFunction(): expected 0 args;
got 1
>>> ageGroupFunction()(21.0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'ScalaUtils$$anonfun$1' object is not
callable
>>> ageGroupFunction().apply(21.0)
2
>>> ageGroupFunction = ageGroupFunction()
```

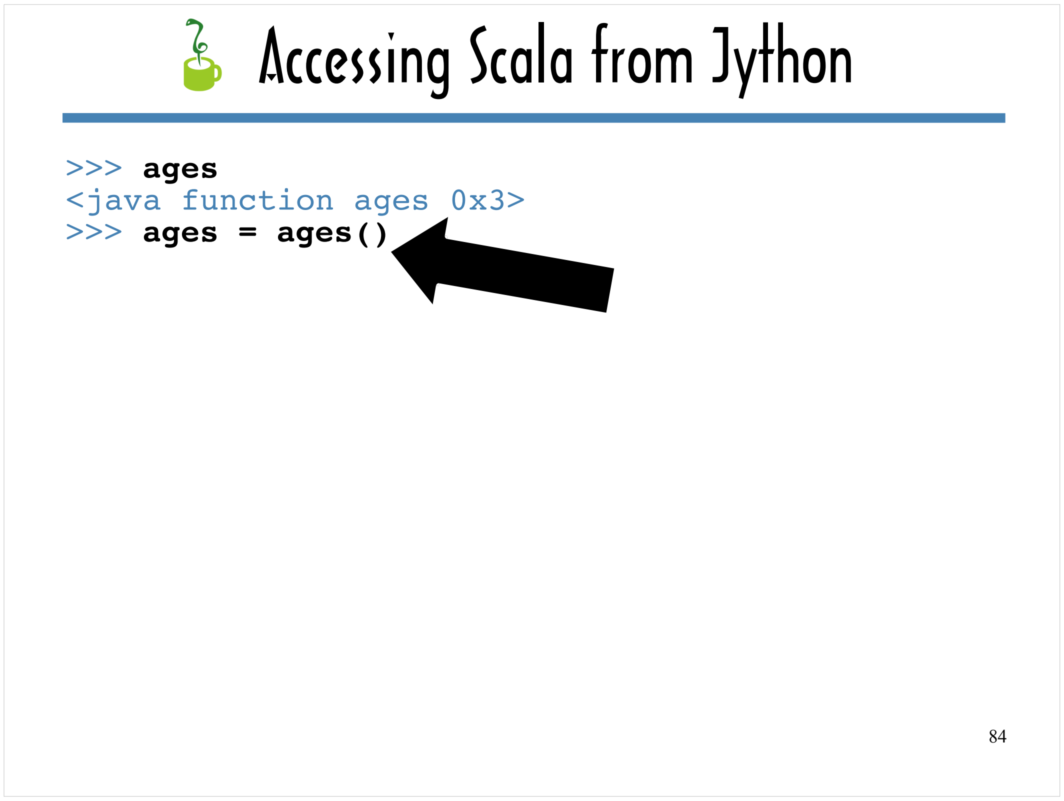
I like to assign the “instantiated” function to a variable with the same name as the function to create the illusion that I'm able to use the function identifier without the parentheses.



## Accessing Scala from Jython

```
>>> ages
<java function ages 0x3>
```

Similarly, you can't call methods on the **`ages`** ListMap without ...



## Accessing Scala from Jython

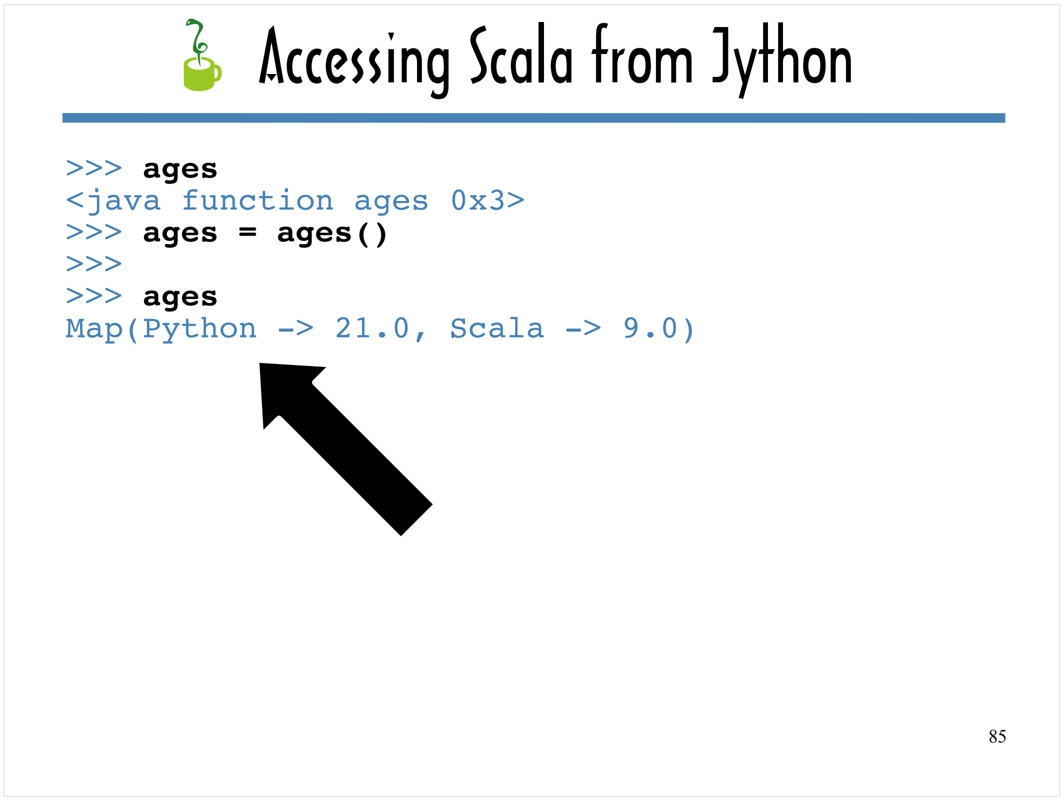
```
>>> ages
<java function ages 0x3>
>>> ages = ages()
```

84

… treating it like a class constructor and following it with parentheses.

The reason for this has to do with the way Scala handles ListMap declarations.

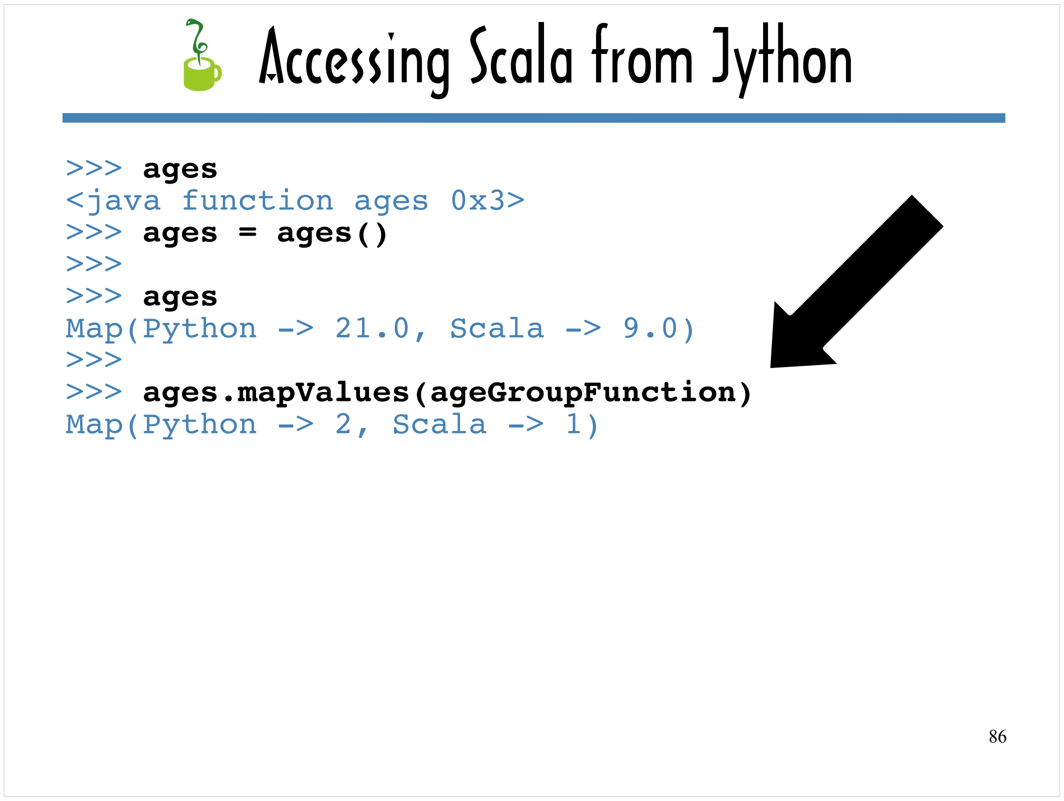
Just as I assigned **`ageGroupFunction()`** to **`AgeGroupFunction`**, I assigned **ages()** to a variable called **`ages`**. This way I can call methods on the **`ages`** ListMap without cluttering up each line of code that references that ListMap with parentheses.



## Accessing Scala from Jython

```
>>> ages
<java function ages 0x3>
>>> ages = ages()
>>>
>>> ages
Map(Python -> 21.0, Scala -> 9.0)
```

You can see here that **`ages`** does, in fact, represents the ListMap defined in the Scala micro-library.

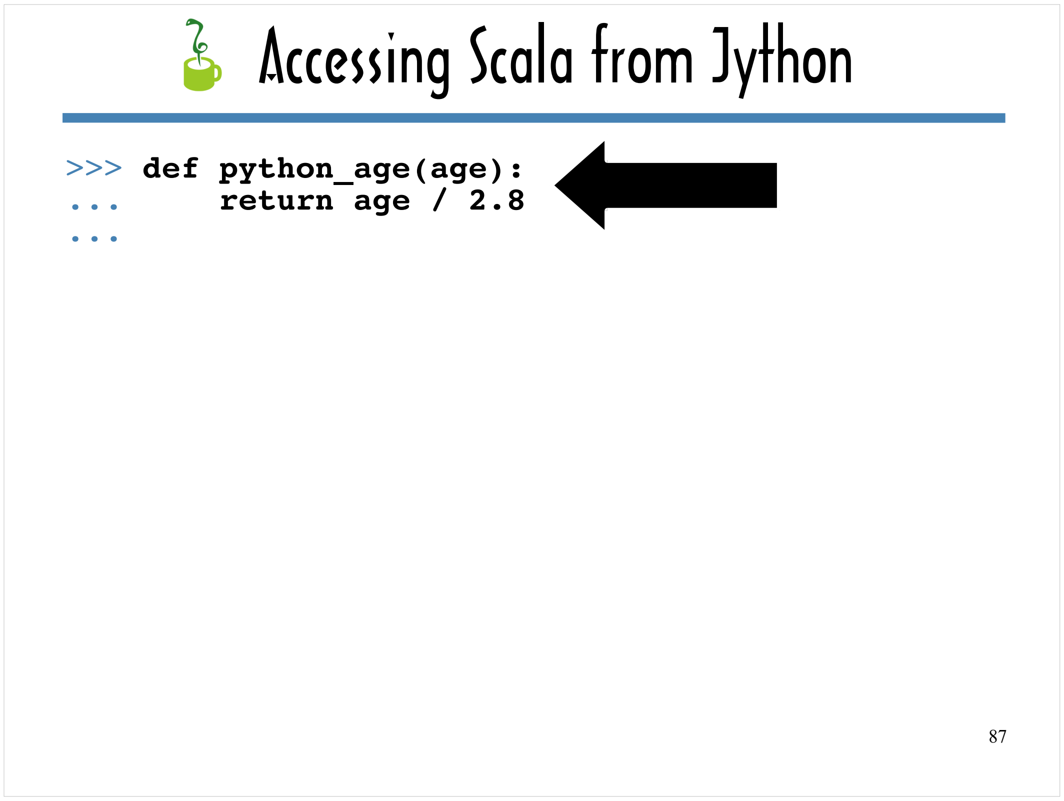


## Accessing Scala from Jython

```
>>> ages
<java function ages 0x3>
>>> ages = ages()
>>>
>>> ages
Map(Python -> 21.0, Scala -> 9.0)
>>>
>>> ages.mapValues(ageGroupFunction)
Map(Python -> 2, Scala -> 1)
```

Here you can see how easy it is to call a function on a collection that was defined in Scala and pass that function a function defined in Scala as an argument – all from within the Jython console.

You can see that if Python and Scala had been runners in the PyCon 5K, and the hypothetical age groups I had defined had been used, Python would have been placed in Age Group #2, while Scala would been placed in Age Group #1.



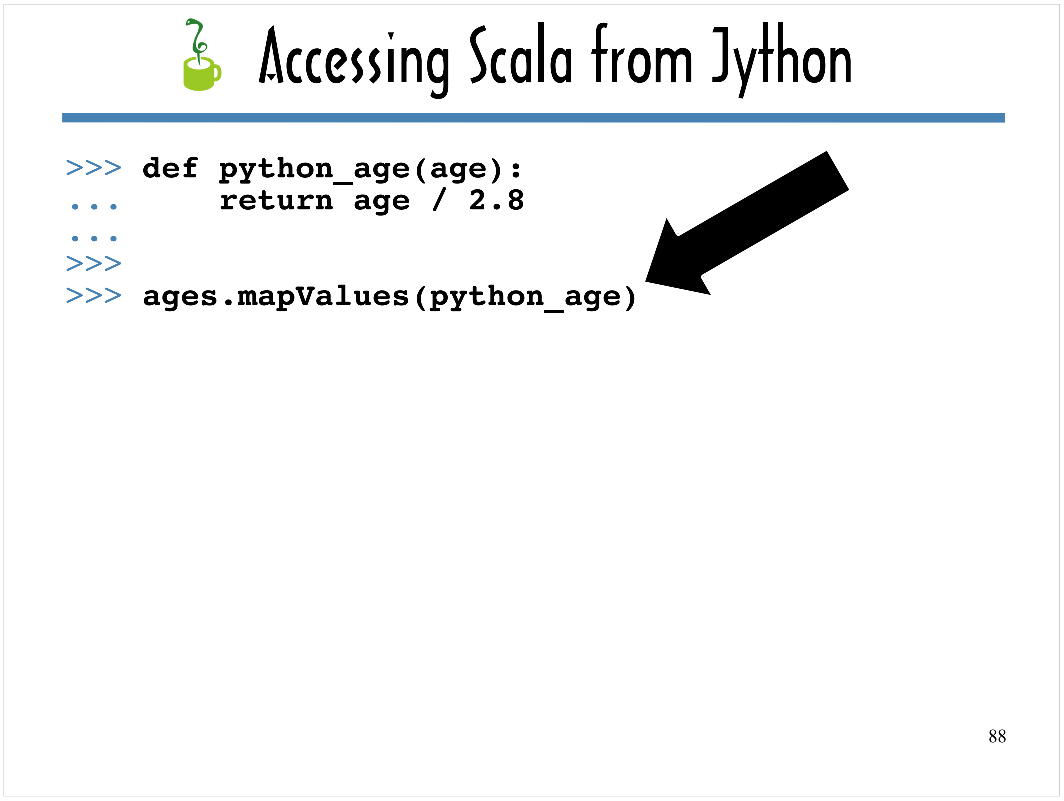
## Accessing Scala from Jython

```
>>> def python_age(age):
...     return age / 2.8
...
```

Now let's see what happens when we try to pass a function defined using Python syntax to **`ListMap#mapValues`**.

In this simulated console, I'm just defining the Jython function I want pass to **`ListMap#mapValues`**,

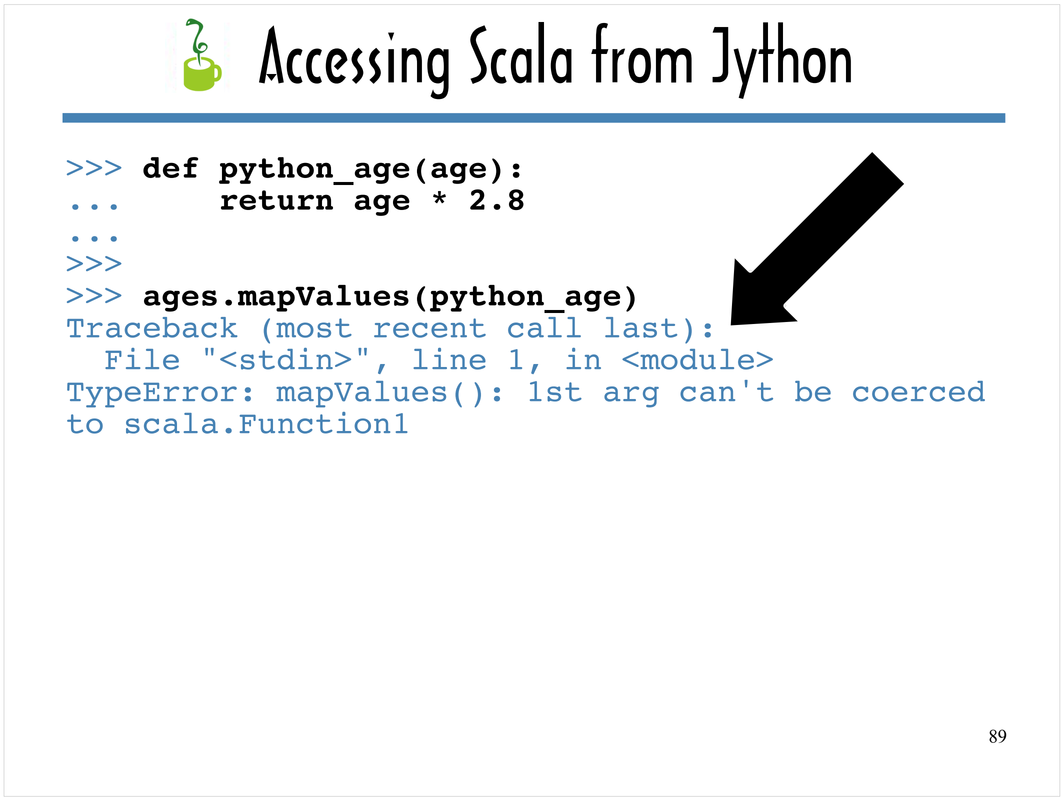
Like the core function in my Python micro-library, it calculates an age in python years.



## Accessing Scala from Jython

```
>>> def python_age(age):
...     return age / 2.8
...
>>>
>>> ages.mapValues(python_age)
```

As it turns out, when I **`pass python_age`** `to` **`ListMap#mapValues`**...

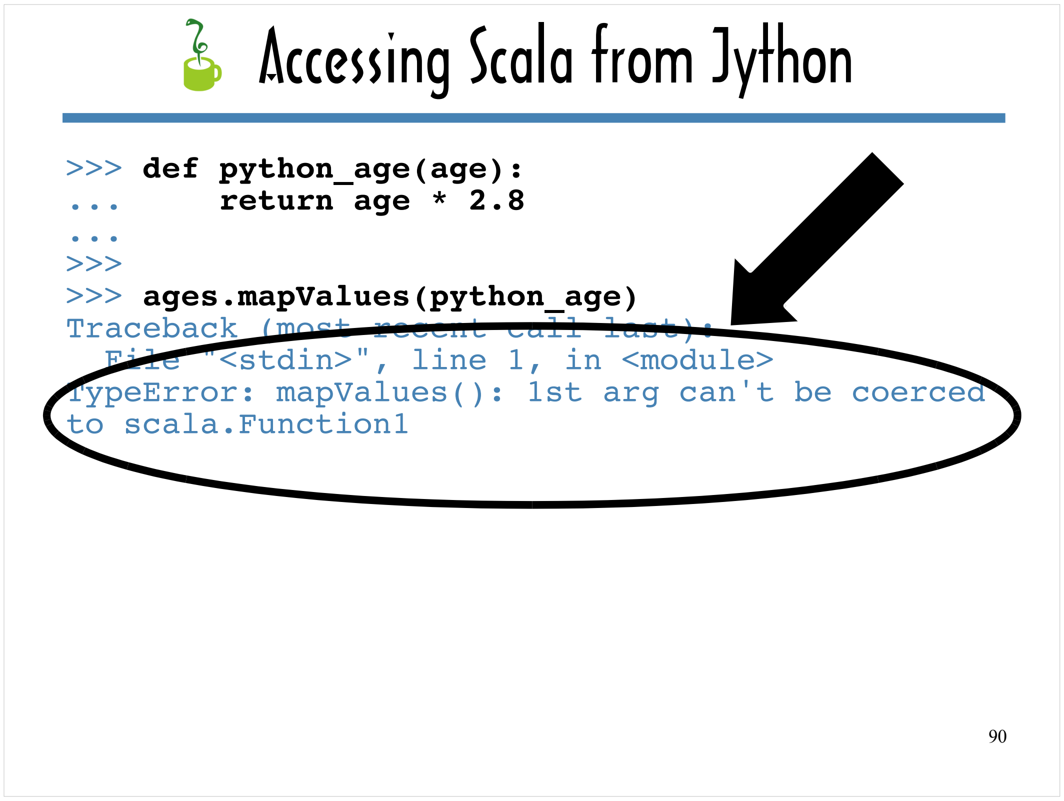


## Accessing Scala from Jython

```
>>> def python_age(age):
...     return age * 2.8
...
>>>
>>> ages.mapValues(python_age)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: mapValues(): 1st arg can't be coerced
to scala.Function1
```

… there are errors in the Jython console.

And if you take a close look at the Jython output ...

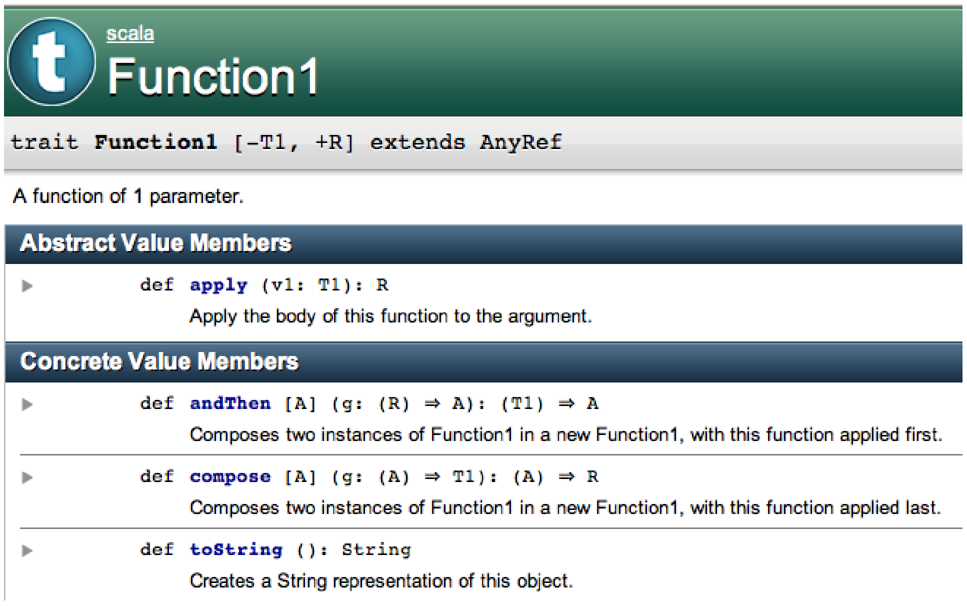


## Accessing Scala from Jython

```
>>> def python_age(age):
...     return age * 2.8
...
>>>
>>> ages.mapValues(python_age)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: mapValues(): 1st arg can't be coerced
to scala.Function1
```

90

… you can see from the TypeError detail that **`ListMap#mapValues`** is expecting something called a Function1, and that **`ListMap#mapValues`** has determined that **`python_age`** does not fit the bill.


scala

Function1

```
trait Function1 [-T1, +R] extends AnyRef
```

A function of 1 parameter.

Abstract Value Members

```
def apply (v1: T1): R
```

Apply the body of this function to the argument.

Concrete Value Members

```
def andThen [A] (g: (R) ⇒ A): (T1) ⇒ A
```

Composes two instances of Function1 in a new Function1, with this function applied first.

```
def compose [A] (g: (A) ⇒ T1): (A) ⇒ R
```

Composes two instances of Function1 in a new Function1, with this function applied last.

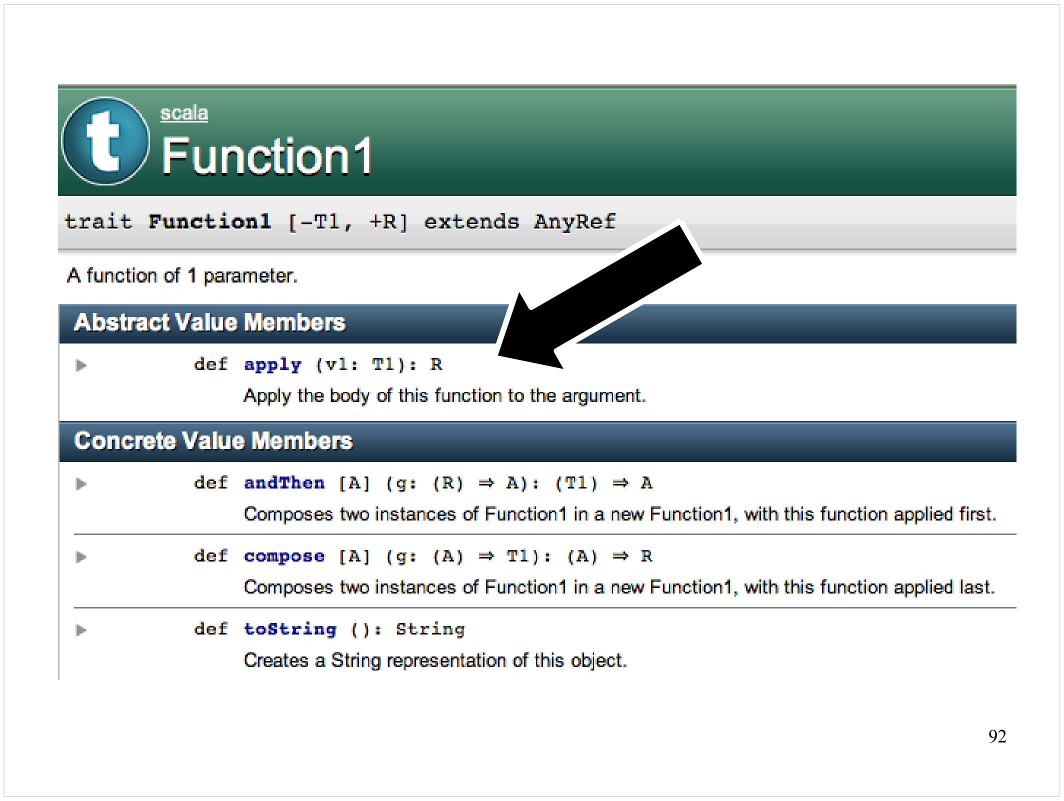
```
def toString (): String
```

Creates a String representation of this object.

Function1 is a Scala trait. Here's a portion of its ScalaDoc.

As I noted earlier, a Scala trait is conceptually related to both abstract base classes and modules in the Python world.

A trait can include declarations for both abstract and concrete methods. A Scala class that mixes in Function1 needs to provide an implementation for ...



scala

Function1

```
trait Function1 [-T1, +R] extends AnyRef
```

A function of 1 parameter.

Abstract Value Members

```
def apply (v1: T1): R
```
Apply the body of this function to the argument.

Concrete Value Members

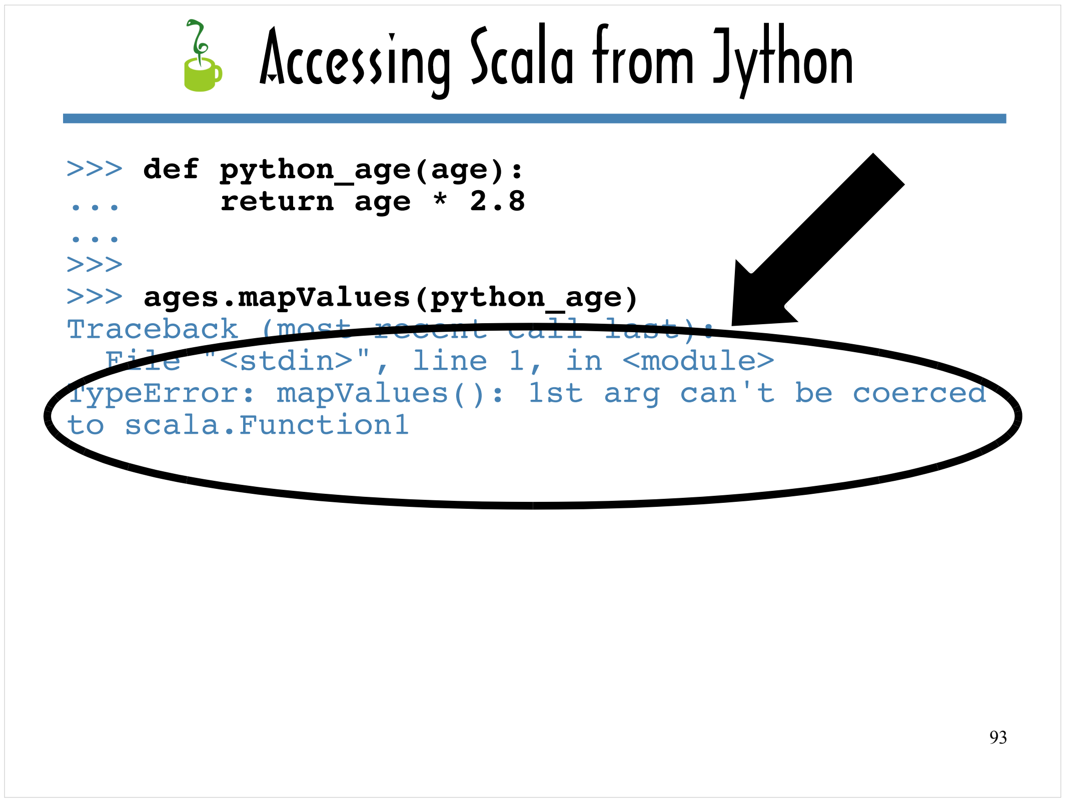
```
def andThen [A] (g: (R) ⇒ A): (T1) ⇒ A
```
Composes two instances of Function1 in a new Function1, with this function applied first.

```
def compose [A] (g: (A) ⇒ T1): (A) ⇒ R
```
Composes two instances of Function1 in a new Function1, with this function applied last.

```
def toString (): String
```
Creates a String representation of this object.

… **`apply`**, Function1's only abstract method.

I explained earlier that when you compile Scala source code that includes a function, the Scala compiler creates a class with an **`apply`** method to represent the function. If a function has one argument, Scala mixes the Function1 trait into the generated class. If a function has two arguments, Scala mixes in the Function2 trait. If a function has three arguments, Scala mixes in the Function3 trait, etc. – up until 22 arguments. The Scala core source includes a Function22 trait. Functions with more than 22 arguments are not supported.

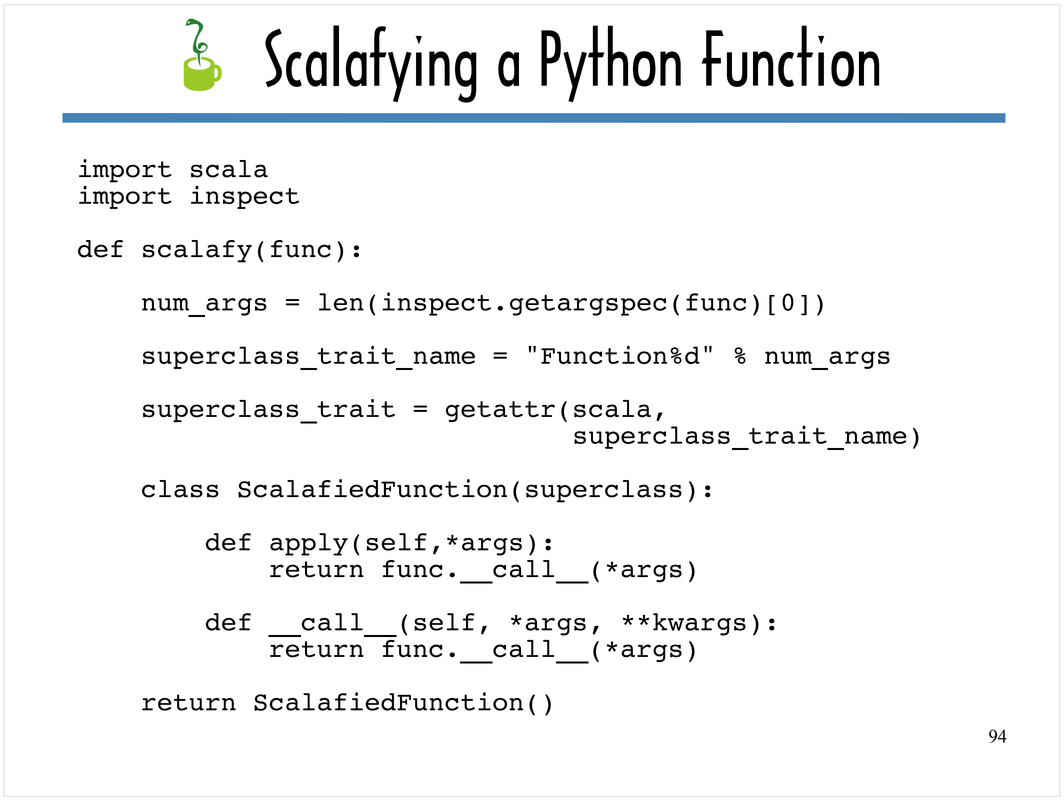


## Accessing Scala from Jython

```
>>> def python_age(age):
...     return age * 2.8
...
>>>
>>> ages.mapValues(python_age)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: mapValues(): 1st arg can't be coerced
to scala.Function1
```

Here's another look at the error the console spewed out.

How can we coerce **`python_age`** to be a Function1 so that **`ListMap#mapValues`** won't choke on it?



## Scalafying a Python Function

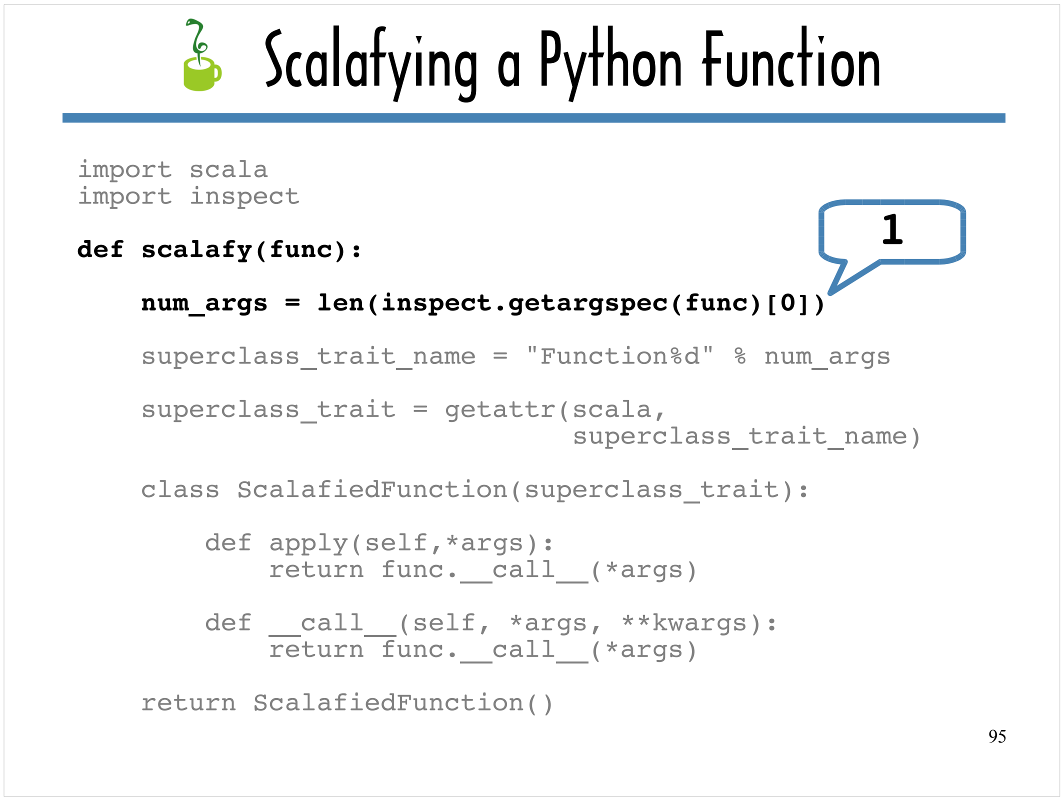
```
import scala
import inspect

def scalafy(func):

    num_args = len(inspect.getargspec(func)[0])

    superclass_trait_name = "Function%d" % num_args

    superclass_trait = getattr(scala,
                               superclass_trait_name)

    class ScalafiedFunction(superclass):

        def apply(self,*args):
            return func.__call__(*args)

        def __call__(self, *args, **kwargs):
            return func.__call__(*args)

    return ScalafiedFunction()
```

94

We can “scalafy” it!

In the next few slides, I'll step through function I wrote to **`scalafy`** a function written using Python syntax.

Where the Scala compiler generates a class with an **`apply`** method to represent a function, **`scalafy`** returns a “scalafied” version of the Python function passed in as the argument, **`func`**.The **`apply`** implementation for the “scalafied” version of **`func`** executes the logic in the body of **`func`**.



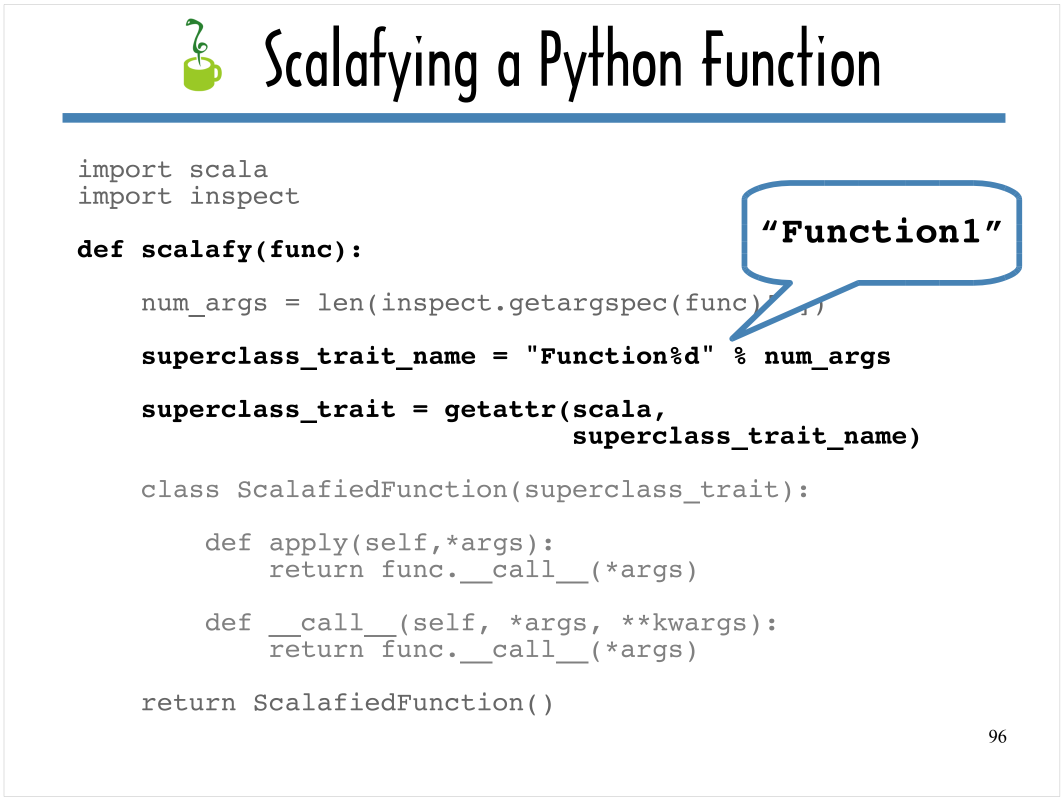
## Scalafying a Python Function

```
import scala
import inspect

def scalafy(func):

    num_args = len(inspect.getargspec(func)[0])

    superclass_trait_name = "Function%d" % num_args

    superclass_trait = getattr(scala,
                               superclass_trait_name)

    class ScalafiedFunction(superclass_trait):

        def apply(self,*args):
            return func.__call__(*args)

        def __call__(self, *args, **kwargs):
            return func.__call__(*args)

    return ScalafiedFunction()
```

“Scalafying” a function written with Python syntax involves subclassing the FunctionN trait with the correct arity.

**`ListMap#mapValues`** will recognize a subclass of Function1 as a Function1.

The first step, highlighted here, is to determine the arity of the function written with Python syntax, using the inspect module.



## Scalafying a Python Function

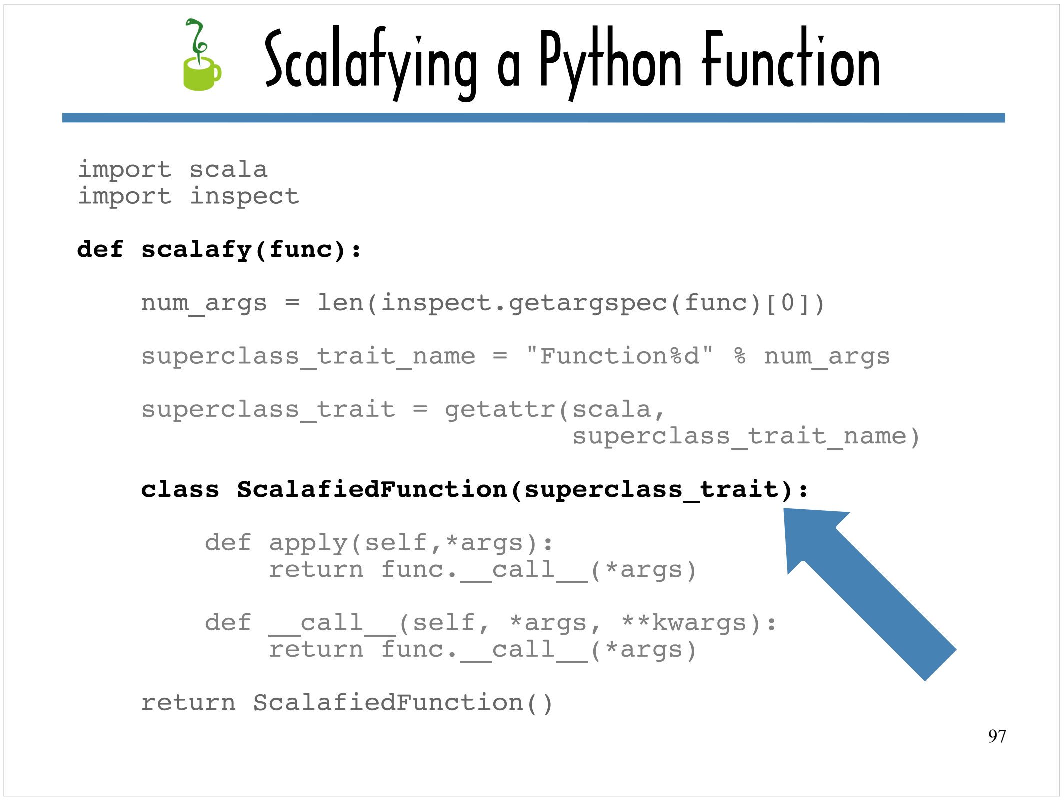
```
import scala
import inspect

def scalafy(func):

    num_args = len(inspect.getargspec(func)[0])

    superclass_trait_name = "Function%d" % num_args

    superclass_trait = getattr(scala,
                               superclass_trait_name)

    class ScalafiedFunction(superclass_trait):

        def apply(self,*args):
            return func.__call__(*args)

        def __call__(self, *args, **kwargs):
            return func.__call__(*args)

    return ScalafiedFunction()
```

**"Function1"**

96

Once the arity is determined, you can easily determine the name of the trait to subclass. In this case, since the arity of **`python_age`** is 1, the trait that needs to be subclassed is "Function1".

Once you have derived the trait name, you can obtain the trait itself using **`getattr`**,

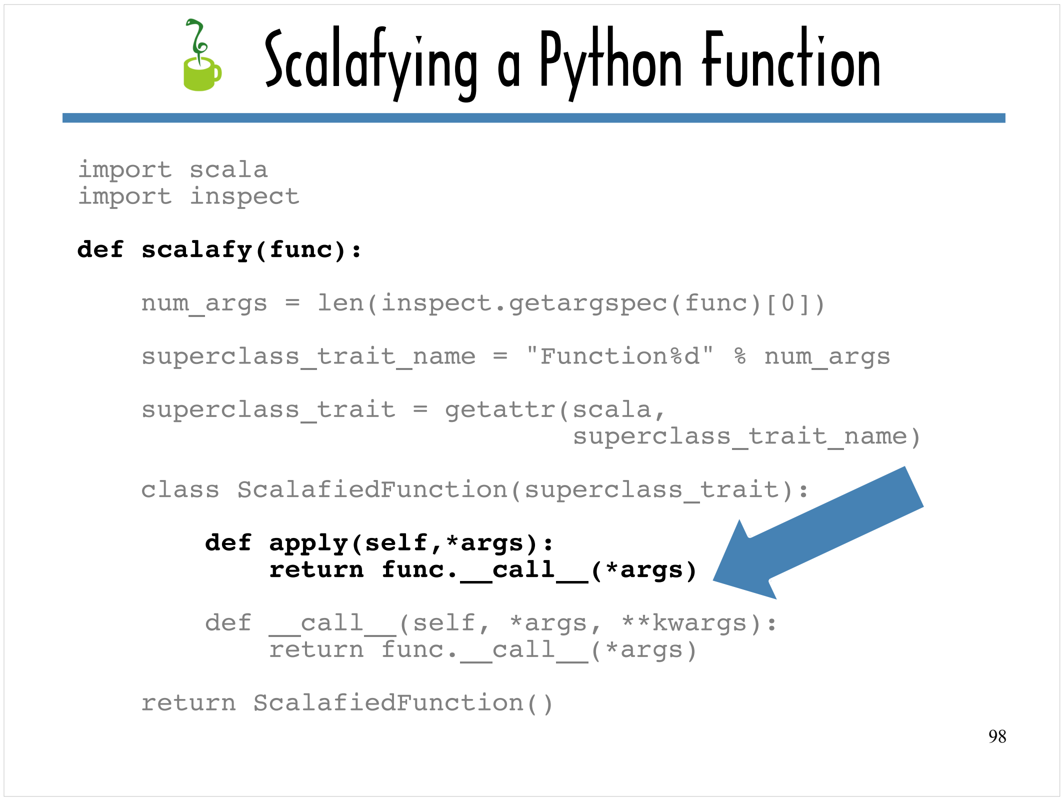


## Scalafying a Python Function

```
import scala
import inspect

def scalafy(func):

    num_args = len(inspect.getargspec(func)[0])

    superclass_trait_name = "Function%d" % num_args

    superclass_trait = getattr(scala,
                               superclass_trait_name)

    class ScalafiedFunction(superclass_trait):

        def apply(self,*args):
            return func.__call__(*args)

        def __call__(self, *args, **kwargs):
            return func.__call__(*args)

    return ScalafiedFunction()
```

97

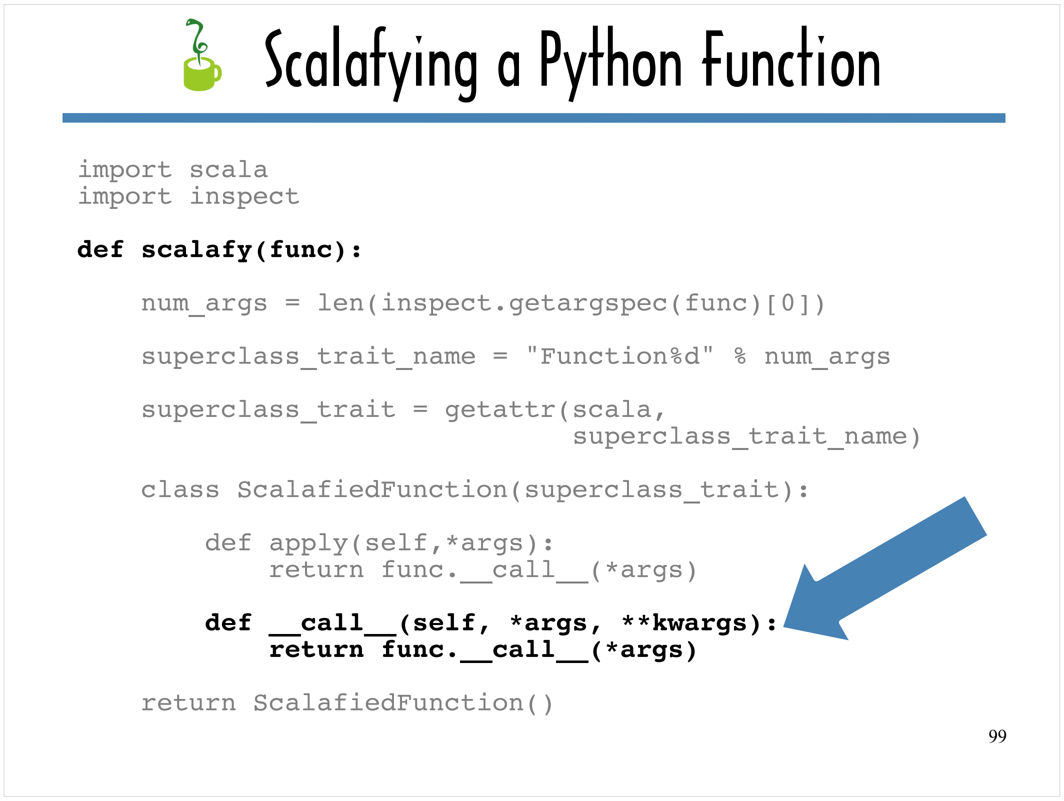
Then the derived FunctionN trait can be used as the superclass in the signature for a local class that provides a concrete implementation of **`apply`**.



## Scalafying a Python Function

```python
import scala
import inspect

def scalafy(func):

    num_args = len(inspect.getargspec(func)[0])

    superclass_trait_name = "Function%d" % num_args

    superclass_trait = getattr(scala,
                               superclass_trait_name)

    class ScalafiedFunction(superclass_trait):

        def apply(self,*args):
            return func.__call__(*args)

        def __call__(self, *args, **kwargs):
            return func.__call__(*args)

    return ScalafiedFunction()
```

The **`apply`** implementationfor the class local to **`scalafy`** uses **`__call__`** to invoke the function passed to **`scalafy`** as its **`func`** argument (which is **`python_age`**, in this case).

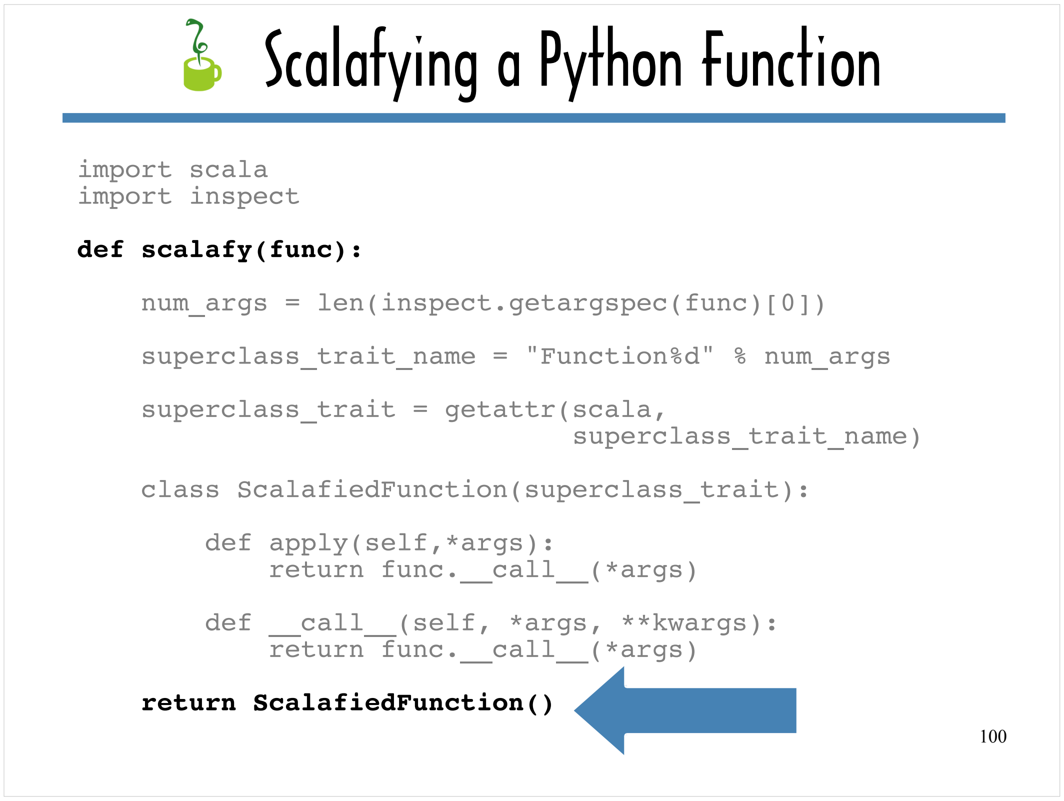


## Scalafying a Python Function

```
import scala
import inspect

def scalafy(func):

    num_args = len(inspect.getargspec(func)[0])

    superclass_trait_name = "Function%d" % num_args

    superclass_trait = getattr(scala,
                               superclass_trait_name)

    class ScalafiedFunction(superclass_trait):

        def apply(self,*args):
            return func.__call__(*args)

        def __call__(self, *args, **kwargs):
            return func.__call__(*args)

    return ScalafiedFunction()
```

The FunctionN subclass that wraps the function passed to **`scalafy`** also needs to implement **`__call__`** – otherwise it will not be recognized as being callable by Jython.

Like the **`apply`** implementation for the FunctionN subclass, the **`__call__`** implementation invokes **`func`**, the function passed to `scalafy` as an argument.

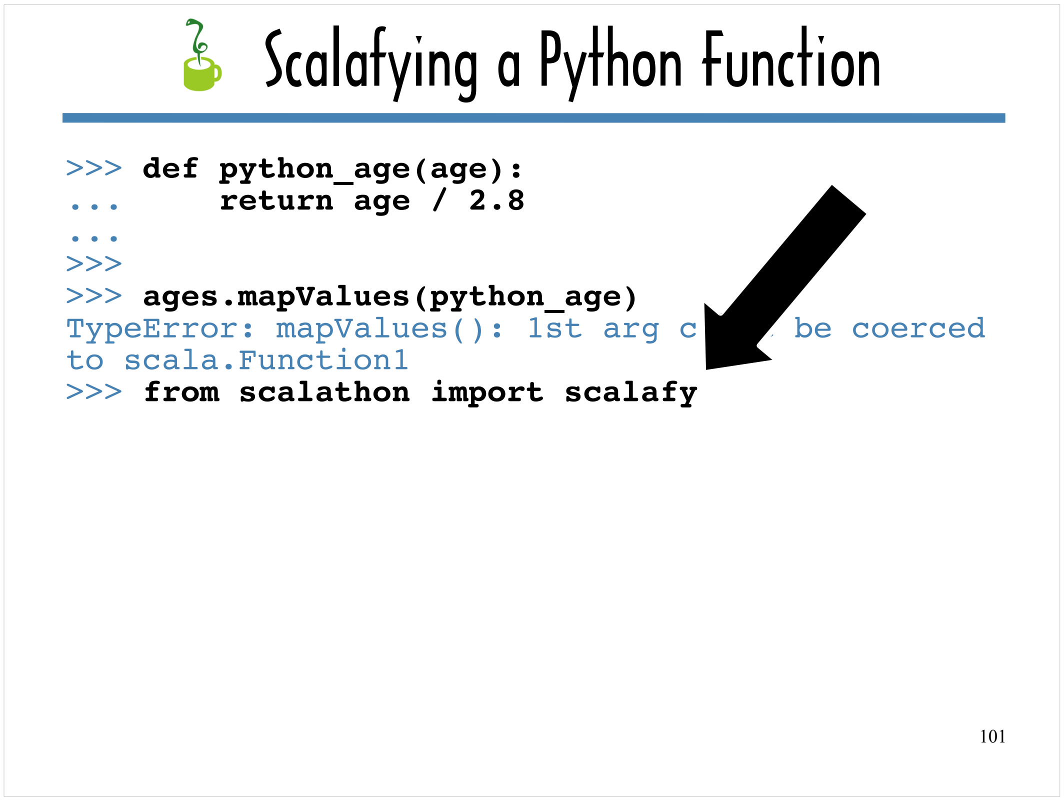


## Scalafying a Python Function

```
import scala
import inspect

def scalafy(func):

    num_args = len(inspect.getargspec(func)[0])

    superclass_trait_name = "Function%d" % num_args

    superclass_trait = getattr(scala,
                               superclass_trait_name)

    class ScalafiedFunction(superclass_trait):

        def apply(self,*args):
            return func.__call__(*args)

        def __call__(self, *args, **kwargs):
            return func.__call__(*args)

    return ScalafiedFunction()
```

100

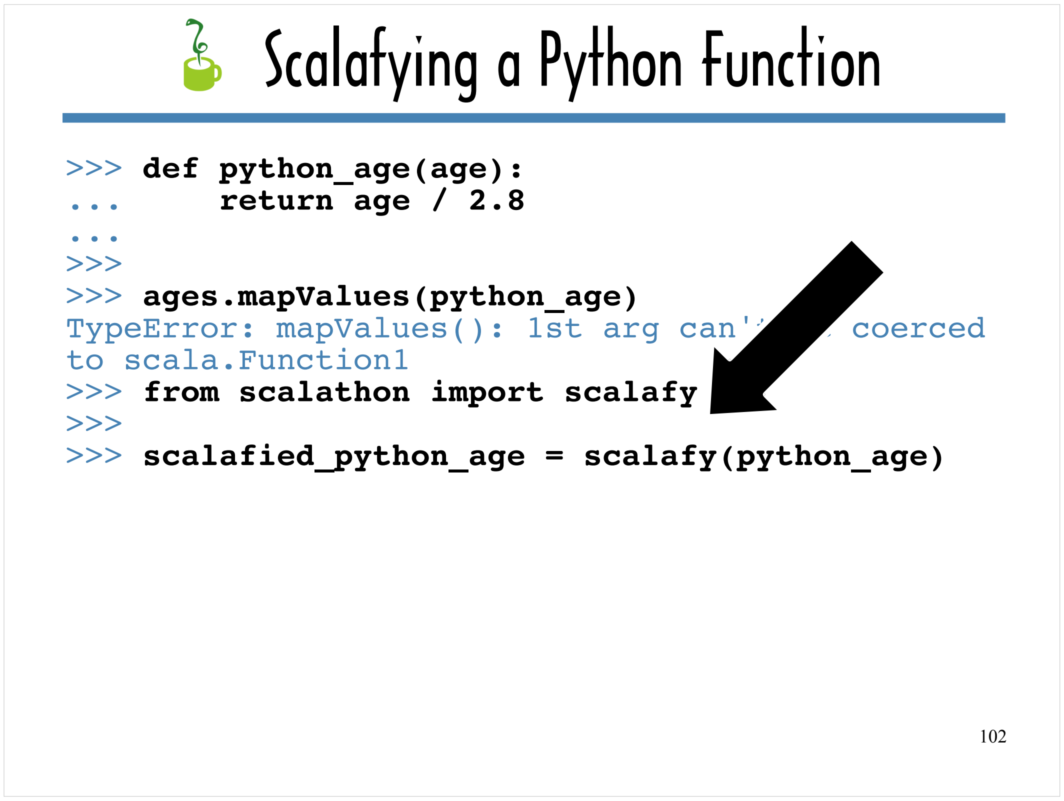
The **`scalafy`** function returns an instance of the class that subclasses the FunctionN trait and implements **`apply`**.



## Scalafying a Python Function

```
>>> def python_age(age):
...     return age / 2.8
...
>>>
>>> ages.mapValues(python_age)
TypeError: mapValues(): 1st arg c          be coerced
to scala.Function1
>>> from scalathon import scalafy
```

If I've named the file containing **`scalafy`** “scalathon.py” (one of the many ways to combine “Scala” and “Python” that I use as module names in this Slide deck), I can import it using the statement I highlighted in this simulated Jython console session.

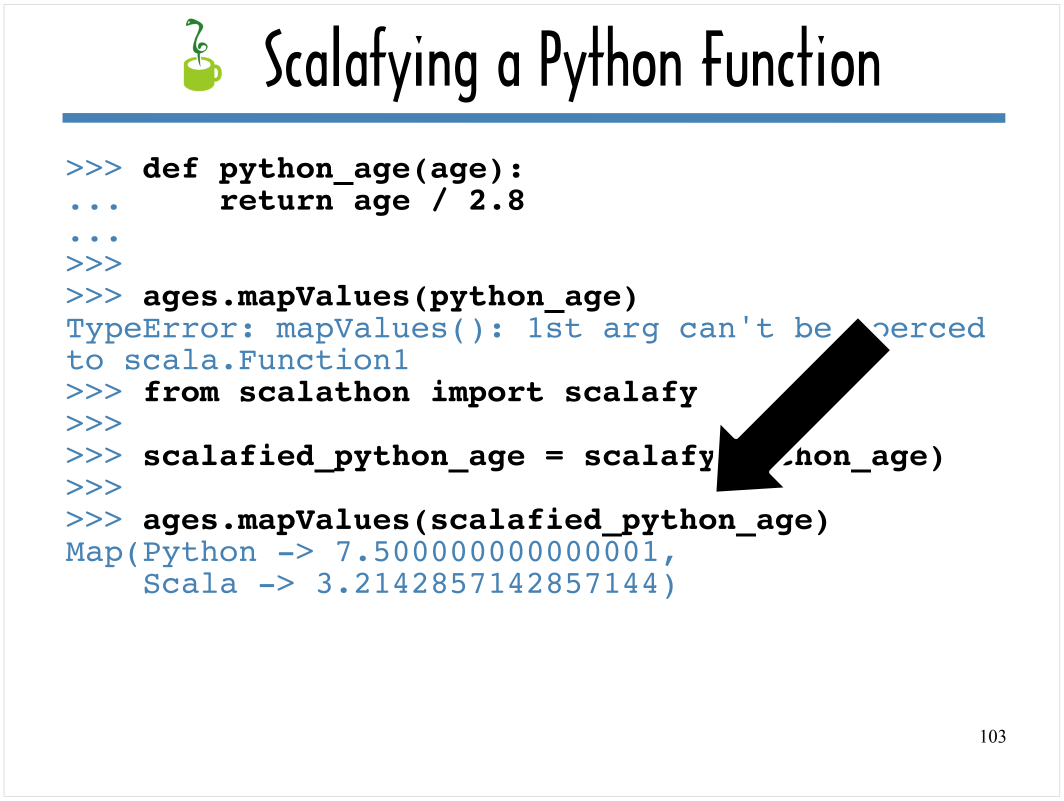


## Scalafying a Python Function

```
>>> def python_age(age):
...     return age / 2.8
...
>>>
>>> ages.mapValues(python_age)
TypeError: mapValues(): 1st arg can't be coerced
to scala.Function1
>>> from scalathon import scalafy
>>>
>>> scalafied_python_age = scalafy(python_age)
```

102

The highlighted code here “scalafies” **`python_age`** by passing it to **`scalafy`** and assigns the “scalafied” version of **`python_age`** to **`scalafied_python_age`**.



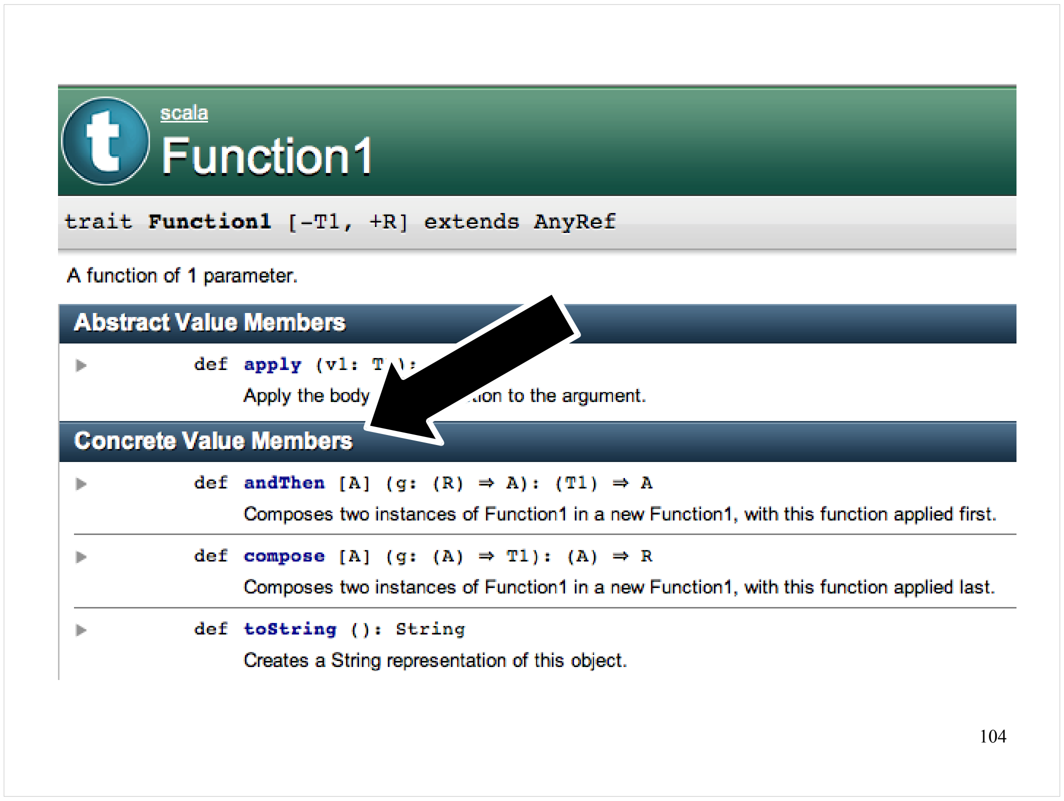
## Scalafying a Python Function

```
>>> def python_age(age):
...     return age / 2.8
...
>>>
>>> ages.mapValues(python_age)
TypeError: mapValues(): 1st arg can't be  oerced
to scala.Function1
>>> from scalathon import scalafy
>>>
>>> scalafied_python_age = scalafy    hon_age)
>>>
>>> ages.mapValues(scalafied_python_age)
Map(Python -> 7.500000000000001,
    Scala -> 3.2142857142857144)
```

As you can see, **`scalafy`** did the trick. Passing **`scalafied_python_age`** to **`ListMap#mapValues`** works.

Instead of a TypeError, the console output shows a ListMap with programming languages names as keys and the ages of the languages – in python years – as the values.

**`ListMap#mapValues`** invokes the function it is passed by calling **`apply`** on that function for each value in the  ListMap.



scala

Function1

```
trait Function1 [-T1, +R] extends AnyRef
```

A function of 1 parameter.

**Abstract Value Members**

```
def apply (v1: T1):
```

Apply the body ... ion to the argument.

**Concrete Value Members**

```
def andThen [A] (g: (R) ⇒ A): (T1) ⇒ A
```

Composes two instances of Function1 in a new Function1, with this function applied first.

```
def compose [A] (g: (A) ⇒ T1): (A) ⇒ R
```

Composes two instances of Function1 in a new Function1, with this function applied last.

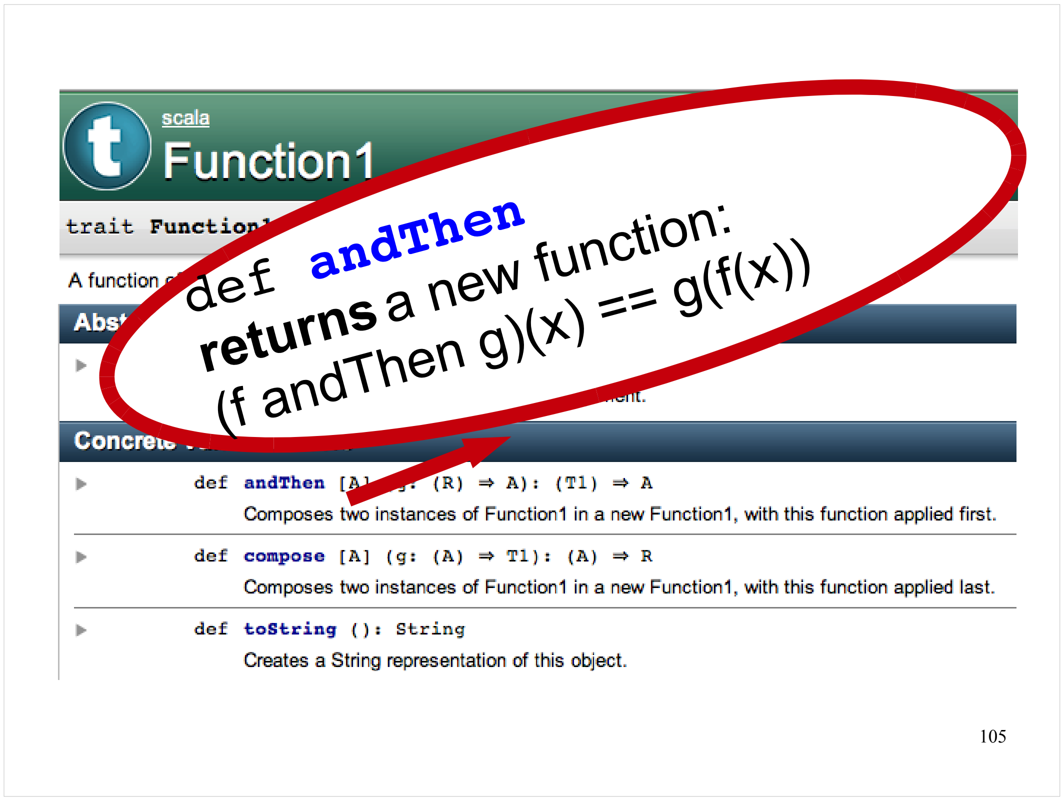
```
def toString (): String
```

Creates a String representation of this object.

**`ListMap#mapValues`** is only interested in the FunctionN's **`apply`** method.

It is not concerned with the methods that the FunctionN
trait provides concrete implementations for: **`andThen`**, **`compose`** and **`ToString`**.

The next set of slides focuses on determining what happens when you pass a function defined using Python syntax to a Scala function that calls one of FunctionN's concrete methods.



scala

Function1

trait Function1

A function

Abst

ment.

Concrete

```
def andThen [A] (g: (R) ⇒ A): (T1) ⇒ A
```
Composes two instances of Function1 in a new Function1, with this function applied first.

```
def compose [A] (g: (A) ⇒ T1): (A) ⇒ R
```
Composes two instances of Function1 in a new Function1, with this function applied last.

```
def toString (): String
```
Creates a String representation of this object.

def **andThen**

**returns** a new function:

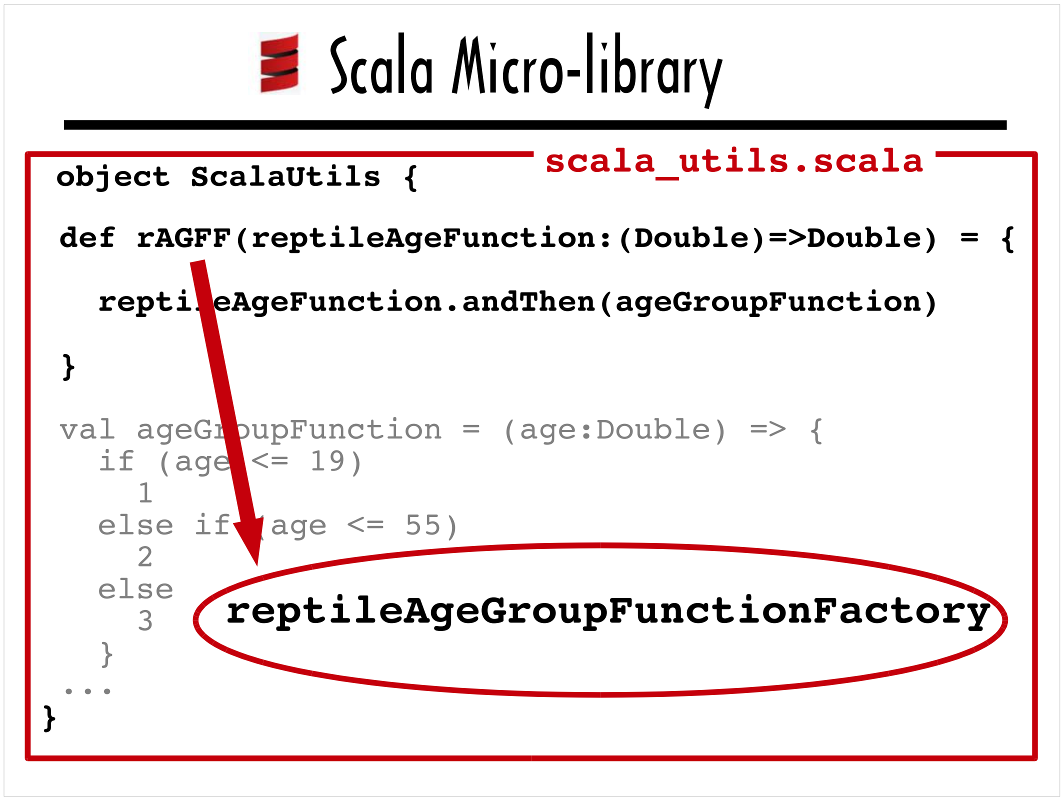
(f andThen g)(x) == g(f(x))

105

**`Function1#andThen`** is the FunctionN concrete method I will use in my examples.

If you call **`Function1#andThen`** on a function called **`f`** and pass in a function called **`g`**, **`Function1#andThen`** will return a new function that takes a single argument does the following
in the following order:
1. invokes **`f`**
2. invokes **`g`** using the return value of **`f`** as the argument to **`g`**

Before I show you what happens when you try to work with Scala code that calls **`Function1#andThen`** in the Jython console, I'll show you how **`Function1#andThen`** works in a sample Scala program.



## Scala Micro-library

scala_utils.scala

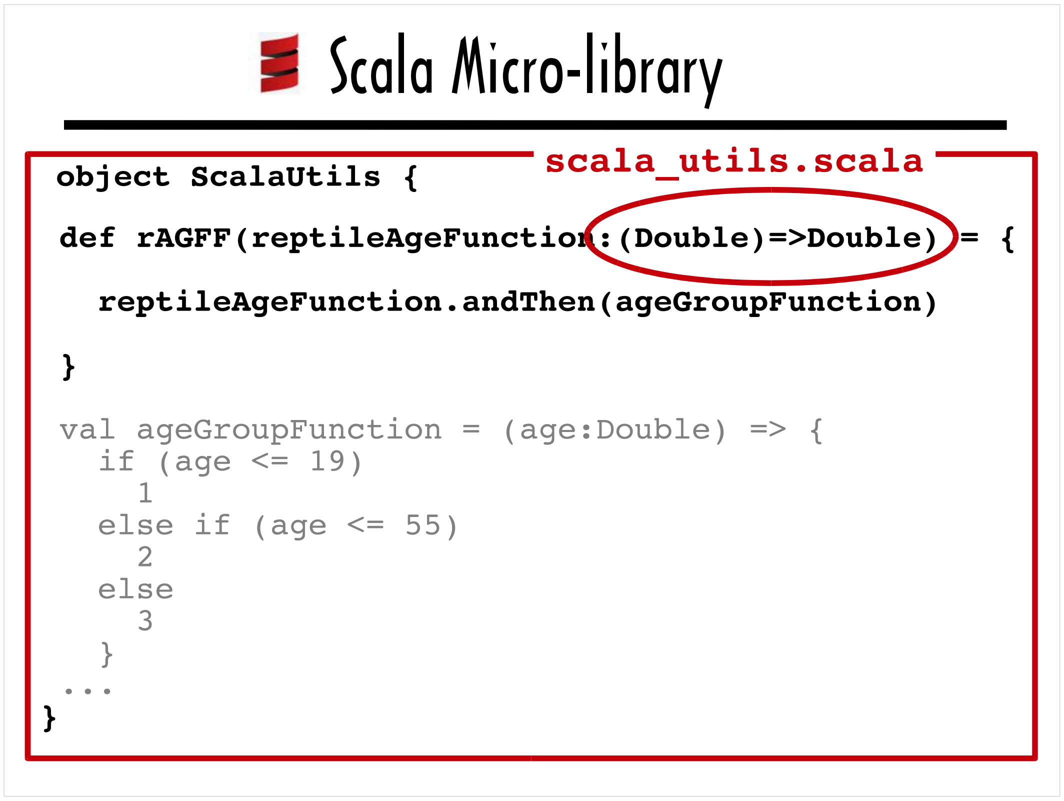
```
object ScalaUtils {

 def rAGFF(reptileAgeFunction:(Double)=>Double) = {

   reptileAgeFunction.andThen(ageGroupFunction)

 }

 val ageGroupFunction = (age:Double) => {
   if (age <= 19)
     1
   else if (age <= 55)
     2
   else
     3
   }
  ...
}
```

Earlier I mentioned that there was an additional method in ScalaUtils that I would cover later in the presentation.

The method is **`reptileAgeGroupFunctionFactory`**, which takes a function as an argument and calls **`Function1#andThen`** on the supplied function.

Because “reptileAgeGroupFunctionFactory” takes up too much space on a slide, I'll use the abbreviation “rAGFF” in the code samples on the slides. In the notes, I'll still use refer to this method as  “reptileAgeGroupFunctionFactory”.

In the next few slides, I'll explain how **`reptileAgeGroupFunctionFactory`** works.

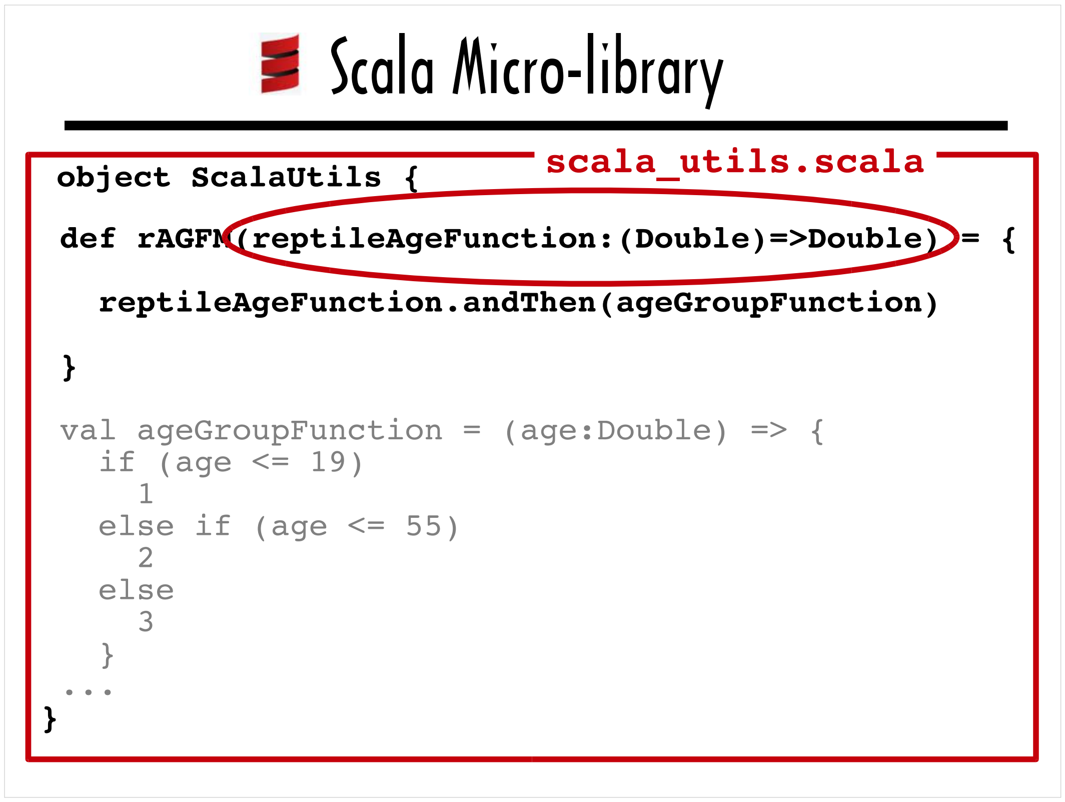


## Scala Micro-library

scala_utils.scala

```
object ScalaUtils {

 def rAGFF(reptileAgeFunction:(Double)=>Double) = {

   reptileAgeFunction.andThen(ageGroupFunction)

 }

 val ageGroupFunction = (age:Double) => {
   if (age <= 19)
     1
   else if (age <= 55)
     2
   else
     3
   }
 ...
}
```

The notation **`(Double)=>Double`** indicates that **`reptileAgeGroupFunctionFactory`** takes a single argument that is a function that takes a Double as an argument and returns a Double.



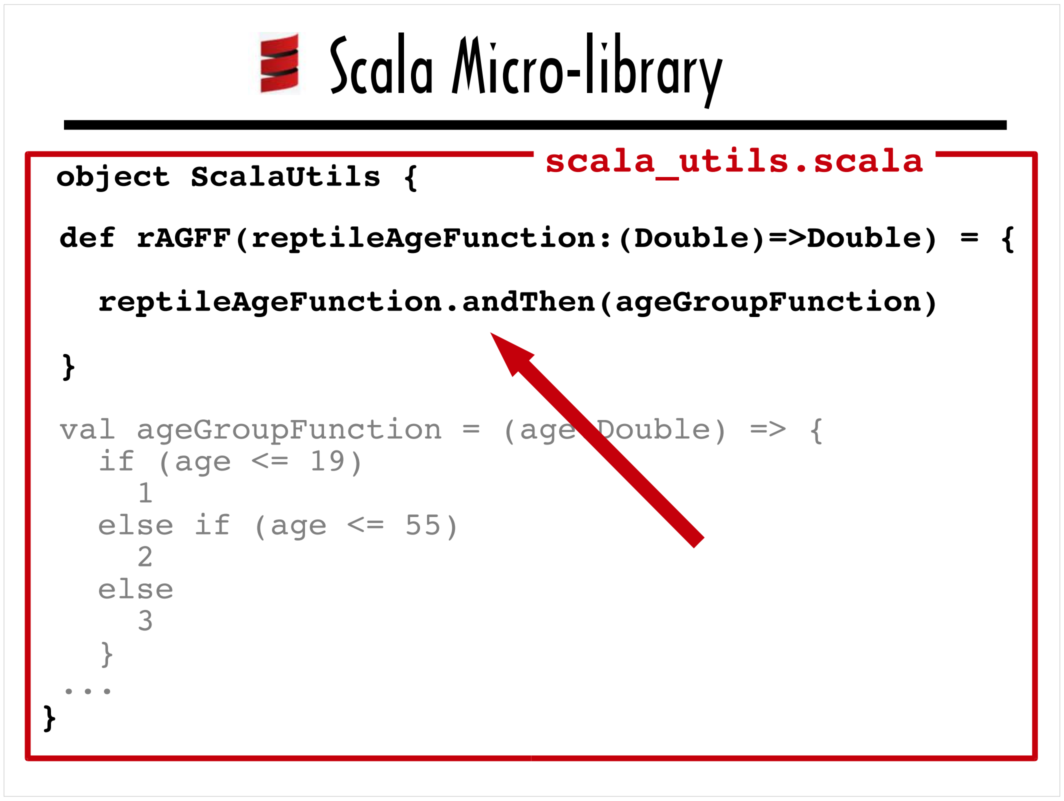
## Scala Micro-library

scala_utils.scala

```
object ScalaUtils {

 def rAGFM(reptileAgeFunction:(Double)=>Double) = {

   reptileAgeFunction.andThen(ageGroupFunction)

 }

 val ageGroupFunction = (age:Double) => {
   if (age <= 19)
     1
   else if (age <= 55)
     2
   else
     3
   }
 ...
}
```

I circled the argument list for **`reptileAgeGroupFunctionFactory`** to point out that its argument (a function that takes a Double and returns a Double) is named **`reptileAgeFunction`**.

The **`reptileAgeGroupFunctionFactory`** method expects to be passed a function like **`python_age`** that takes an age as an argument and converts the age based on the average life span of a particular reptile.



## Scala Micro-library

scala_utils.scala

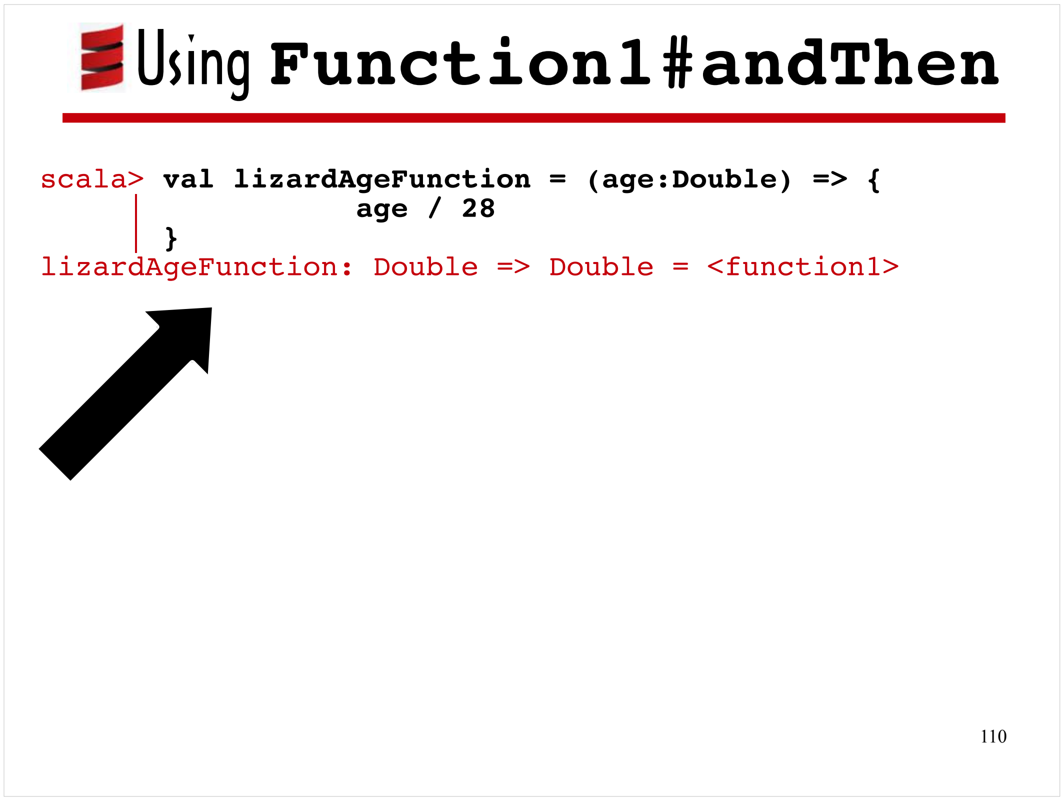
```
object ScalaUtils {

 def rAGFF(reptileAgeFunction:(Double)=>Double) = {

   reptileAgeFunction.andThen(ageGroupFunction)

 }

 val ageGroupFunction = (age:Double) => {
   if (age <= 19)
     1
   else if (age <= 55)
     2
   else
     3
   }
 ...
}
```

The **reptileAgeGroupFunctionFactory** method calls **Function1#andThen** on **reptileAgeFunction** using **ageGroupFunction** as the argument to **Function1#andThen**.

The expression **reptileAgeFunction.andThen(ageGroupFunction)** returns as new function that does the following, in order:
1. invokes **reptileAgeFunction**
2. invokes **ageGroupFunction**, using the return value of **reptileAgeFunction** as the argument to **ageGroupFunction**

So, if you were to pass **python_age** to **reptileAgeGroupFunctionFactory**, a new function would be created that would:
1. invoke **python_age** to determine the equivalent of the age passed in as an argument in python years
2. invoke **ageGroupFunction**, using the python years equivalent of the age as the argument to **ageGroupFunction**

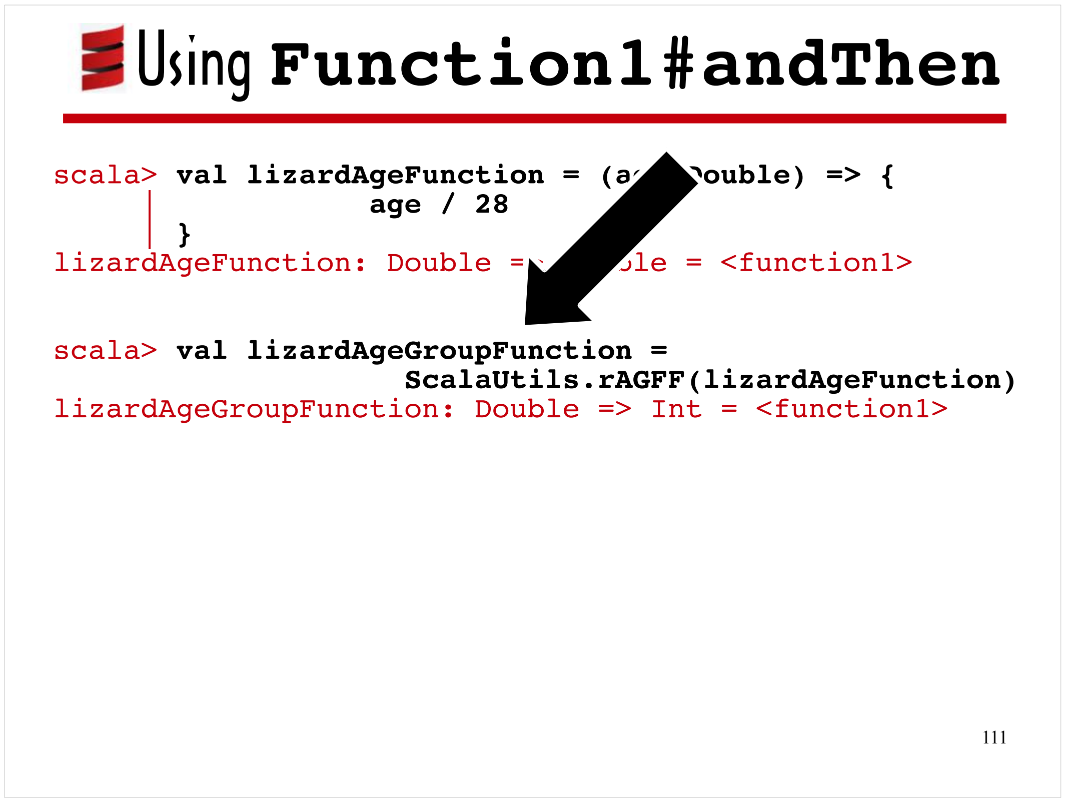
Shortly I'll show you what happens when you try to pass the “scalafied” version of **python_age** to **reptileAgeGroupFunctionFactory**. First I'll show you a simulated Scala console session that features **ReptileAgeGroupFunctionFactory** so you can get a feel for how **Function1#andThen** works in Scala.



## Using Function1#andThen

```
scala> val lizardAgeFunction = (age:Double) => {
     |                age / 28
     | }
lizardAgeFunction: Double => Double = <function1>
```

First I defined a Scala function that calculates a person's age in lizard years.



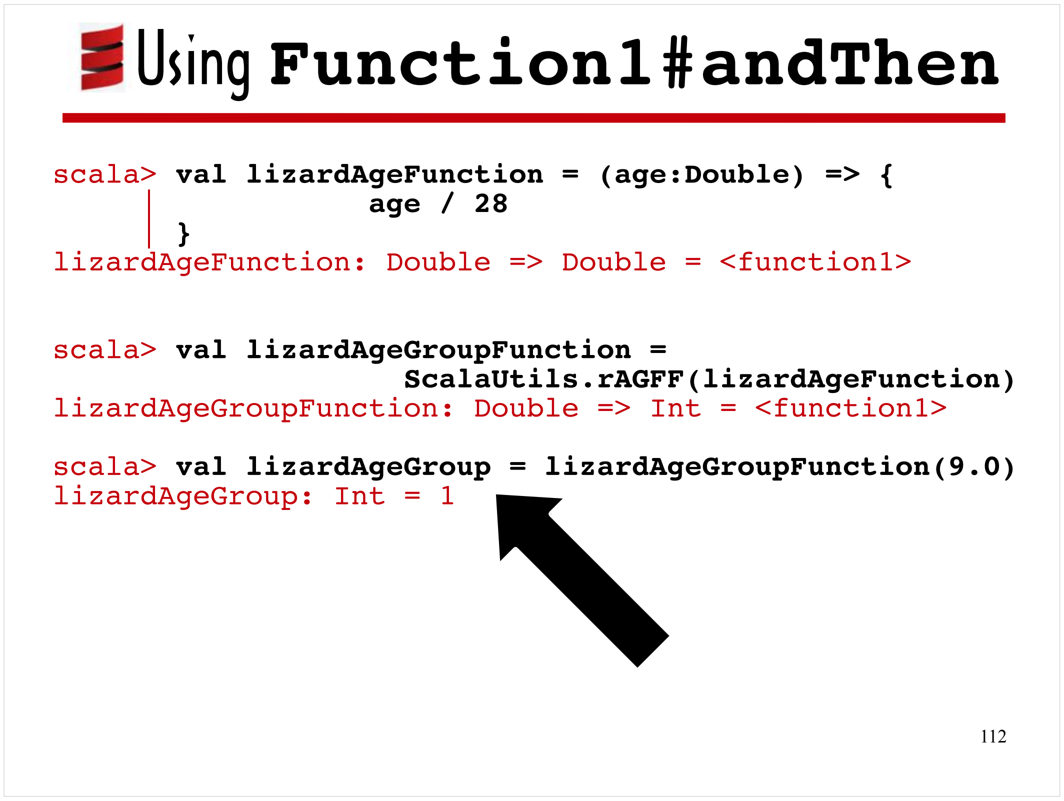
## Using Function1#andThen

```
scala> val lizardAgeFunction = (a    ouble) => {
                      age / 28
                }
lizardAgeFunction: Double =       ble = <function1>


scala> val lizardAgeGroupFunction =
                        ScalaUtils.rAGFF(lizardAgeFunction)
lizardAgeGroupFunction: Double => Int = <function1>
```

111

Then I pass it to **`ScalaUtils#reptileAgeGroupFunctionFactory`** to create a new function that I assign to the variable **`lizardAgeGroupFunction`**.

Notice that the console reports that the type of **`lizardAgeGroupFunction`** is: **`Double => Int`**, which is the notation for a function that takes a Double and returns an Int. This is the type you would expect for a function that takes an age as a parameter and returns the age group number for the lizard years equivalent of that age.



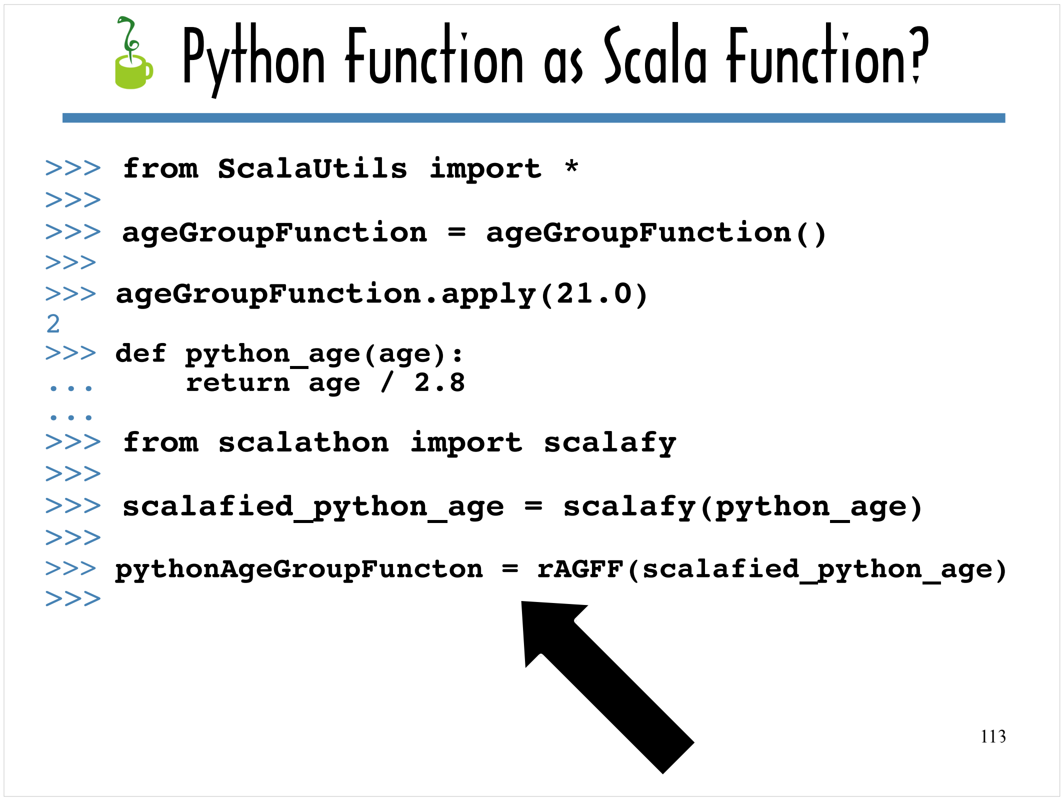
## Using Function1#andThen

```
scala> val lizardAgeFunction = (age:Double) => {
     |                 age / 28
     | }
lizardAgeFunction: Double => Double = <function1>


scala> val lizardAgeGroupFunction =
                    ScalaUtils.rAGFF(lizardAgeFunction)
lizardAgeGroupFunction: Double => Int = <function1>

scala> val lizardAgeGroup = lizardAgeGroupFunction(9.0)
lizardAgeGroup: Int = 1
```

112

Finally, I use the newly minted function, **`lizardAgeGroupFunction`** to determine which age group 9-year-old Scala would fall into if it was a lizard, The result is 1.

Now I'll show you what happens when I try passing a function defined using Python syntax to **`reptileAgeGroupFunctionFactory`** in a simulated Jython console session.



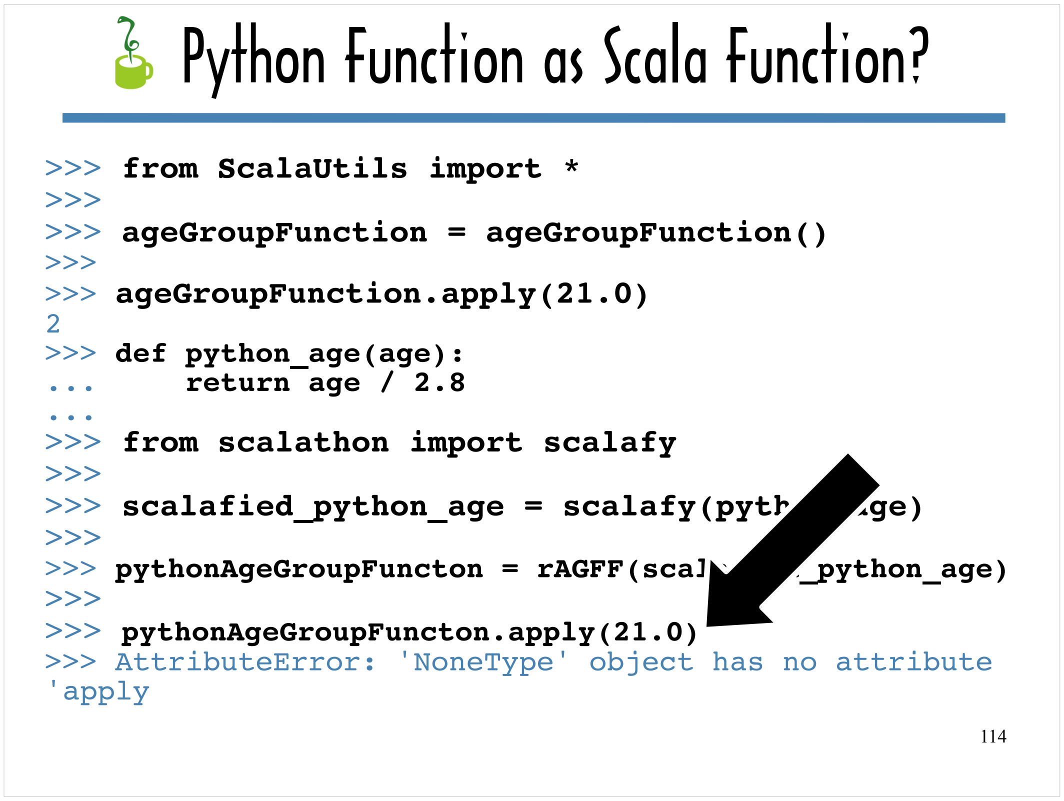
## Python Function as Scala Function?

```
>>> from ScalaUtils import *
>>>
>>> ageGroupFunction = ageGroupFunction()
>>>
>>> ageGroupFunction.apply(21.0)
2
>>> def python_age(age):
...     return age / 2.8
...
>>> from scalathon import scalafy
>>>
>>> scalafied_python_age = scalafy(python_age)
>>>
>>> pythonAgeGroupFuncton = rAGFF(scalafied_python_age)
>>>
```

113

So far so, good. There are no errors in the console output when I pass the “scalafied” version of **`python_age`** to **`reptileAgeGroupFunctionFactory`**, to create **`pythonAgeGroupFunction`**,

But...



## Python Function as Scala Function?

```
>>> from ScalaUtils import *
>>>
>>> ageGroupFunction = ageGroupFunction()
>>>
>>> ageGroupFunction.apply(21.0)
2
>>> def python_age(age):
...     return age / 2.8
...
>>> from scalathon import scalafy
>>>
>>> scalafied_python_age = scalafy(pyth   ge)
>>>
>>> pythonAgeGroupFuncton = rAGFF(scal      _python_age)
>>>
>>> pythonAgeGroupFuncton.apply(21.0)
>>> AttributeError: 'NoneType' object has no attribute
'apply
```

114

… when I try to invoke **apply** on **pythonAgeGroupFunction** (recall that **reptileAgeGroupFunctionFactory** returns a Scala function, and that you need to call a Scala function's **apply** method in order to execute the function logic from within Jython), the console displays error details.

The console indicates that **pythonAgeGroupFunction** is not a Scala function, but rather that its type is "None".

There's only one line in the method body for **reptileAgeGroupFunctionFactory**, and it invokes **Function1#andThen** on The function passed in as an argument (in this case **scalafied_python_age**). So Clearly there's a problem with the **Function1#andThen** implementation on **scalafied_python_age**.

You may well be wondering how there could be a problem with **scalafied_python_age**'s implementation of **Function1#andThen** if **scalafied_python_age** subclasses Function1 and the Function1 definition includes a concrete implementation of **Function1#andThen**.

## Scalafying a Python Function

Code Commit Blog

Integrating Scala into JRuby

Daniel Spiewak

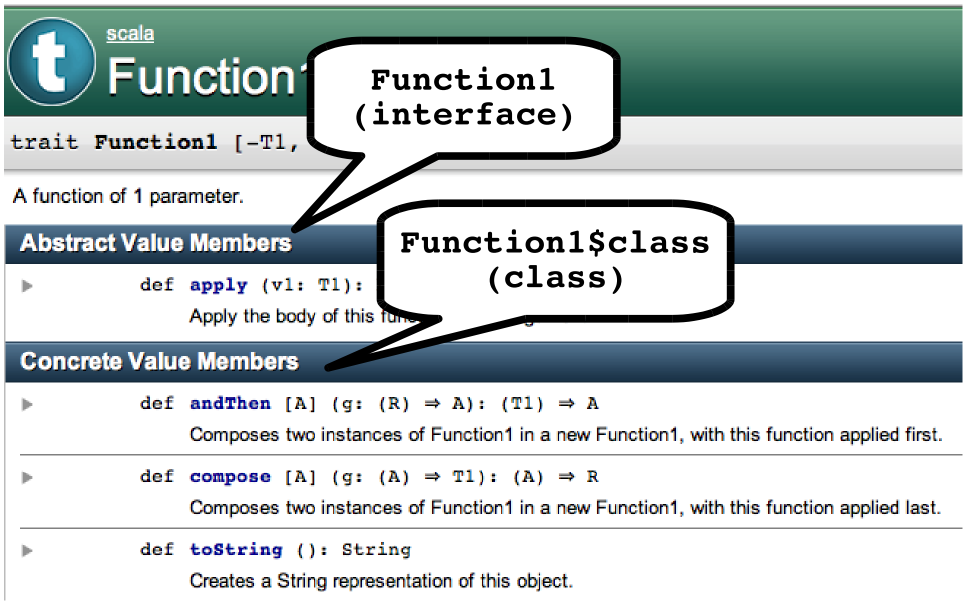
http://www.codecommit.com/blog/ruby/integrating-scala-into-jruby

Although it's about JRuby/Scala integration and not Jython, Daniel Spiewak's blog post “Integrating Scala into JRuby” is very relevant as we consider the question: What's wrong with **`scalafied_python_age`**'s implementation of **`Function1#andThen`**?

This blog post explains how Scala finesses Java's constructs in order to support its trait semantics.

There's no concept of a trait in Java, so the Scala compiler generates a Java interface that includes the trait's abstract method declarations and a Java class that includes the trait's concrete method definitions.

A Java interface is similar to a Scala trait (or a Python abstract base class), but it can only include abstract method declarations.



```
trait Function1 [-T1,
```

A function of 1 parameter.

```
def apply (v1: T1):
```

Apply the body of this fun

```
def andThen [A] (g: (R) ⇒ A): (T1) ⇒ A
```

Composes two instances of Function1 in a new Function1, with this function applied first.

```
def compose [A] (g: (A) ⇒ T1): (A) ⇒ R
```

Composes two instances of Function1 in a new Function1, with this function applied last.

```
def toString (): String
```

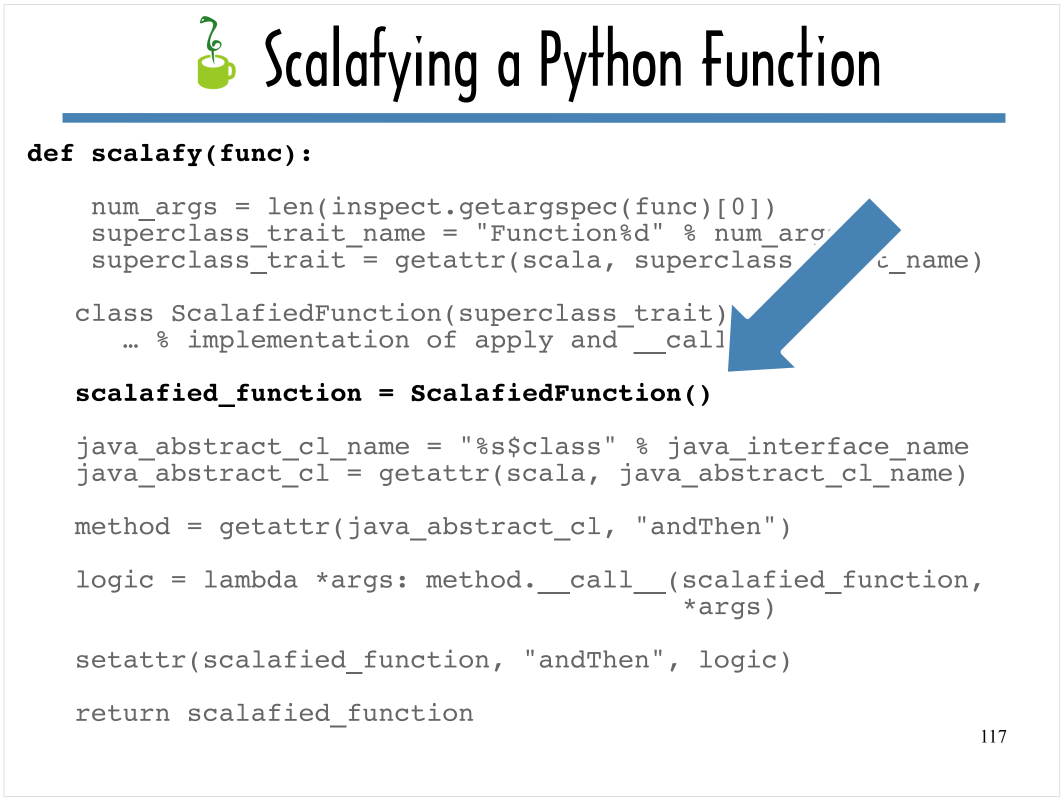
Creates a String representation of this object.

The Scala compiler gives the Java interface the same name as the trait (in this case, "Function1") and the generated class name is the trait name with "$class" appended to It (include this case, Function1$class.

When a Scala class mixes in a trait, the Scala compiler generates a method definition that invokes the corresponding method on the generated <FunctionN>$class – for each concrete method defined in the trait.

Of course, Jython is Java-based, not Scala-based.  Jython has access to the compiled Scala, not the Scala source. So to Jython, "Function1" is a Java interface, not a trait. When **`scalafied_python_age`** subclasses "Function1" and Jython's programatic compiler generates Java bytecode to represent **scalafied_python_age**, Jython does not know that **scalafied_python_age**'s **`Function1#andThen`** implementation needs to invoke the **`andThen`** implementation on a class called "Function1$class".

Faced with the similar problem, but with a JRuby class that subclasses Function1, Spiewak suggests that in order to make a JRuby-based subclass of Function1 behave like a Scala function, you need to derive the name of the "<FunctionN>$class" that contains the trait's concrete implementations, programatically capture the logic for each concrete implementation, and dynamically generate implementations of the trait's concrete methods that invoke that captured logic.



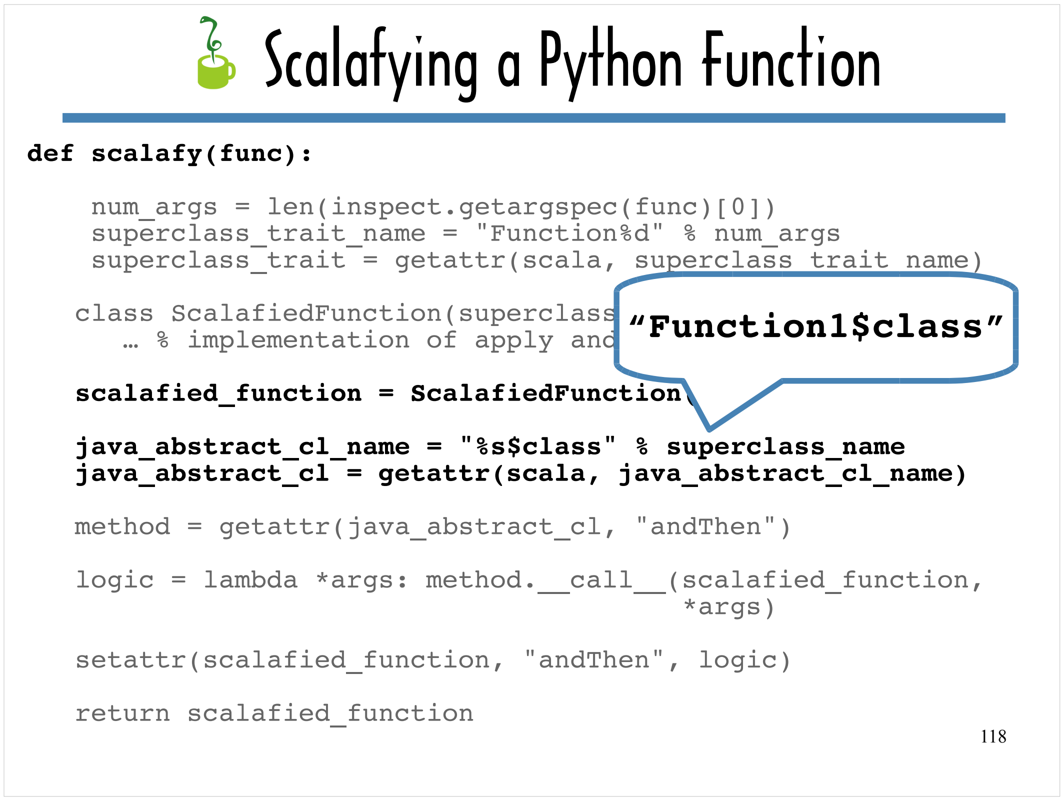
## Scalafying a Python Function

```
def scalafy(func):

    num_args = len(inspect.getargspec(func)[0])
    superclass_trait_name = "Function%d" % num_arg
    superclass_trait = getattr(scala, superclass     c_name)

   class ScalafiedFunction(superclass_trait)
      … % implementation of apply and __call

   scalafied_function = ScalafiedFunction()

   java_abstract_cl_name = "%s$class" % java_interface_name
   java_abstract_cl = getattr(scala, java_abstract_cl_name)

   method = getattr(java_abstract_cl, "andThen")

   logic = lambda *args: method.__call__(scalafied_function,
                                         *args)

   setattr(scalafied_function, "andThen", logic)

   return scalafied_function
```

117

I modified **`scalafy`** based on Spiewak's blog entry.

Instead of returning an instance of ScalafiedFunction following the class definition, I assign the newly minted ScalafiedFunction to **`scalafied_function`**, ...

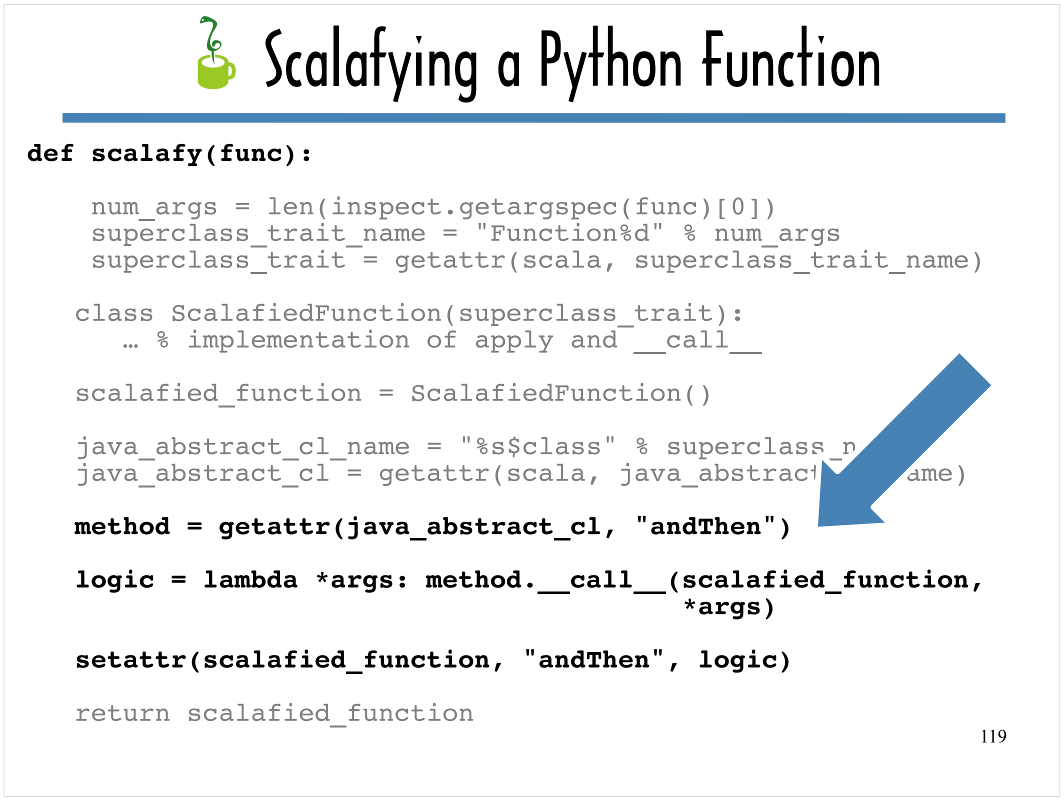


## Scalafying a Python Function

```
def scalafy(func):

    num_args = len(inspect.getargspec(func)[0])
    superclass_trait_name = "Function%d" % num_args
    superclass_trait = getattr(scala, superclass_trait_name)

   class ScalafiedFunction(superclass
      … % implementation of apply and

   scalafied_function = ScalafiedFunction(

   java_abstract_cl_name = "%s$class" % superclass_name
   java_abstract_cl = getattr(scala, java_abstract_cl_name)

   method = getattr(java_abstract_cl, "andThen")

   logic = lambda *args: method.__call__(scalafied_function,
                                         *args)

   setattr(scalafied_function, "andThen", logic)

   return scalafied_function
```

118

… derive the “<FunctionN>$class” based on the arity of the function defined using Python syntax, which is “Function1$class” in this case and then …

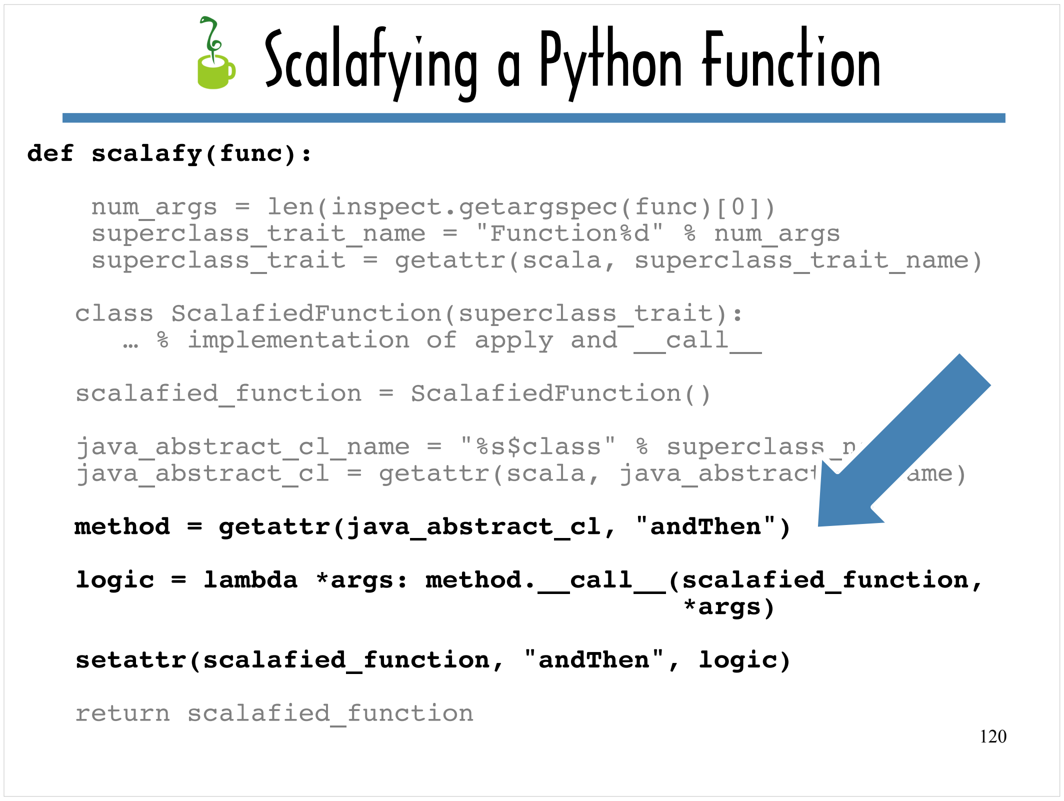


## Scalafying a Python Function

```
def scalafy(func):

    num_args = len(inspect.getargspec(func)[0])
    superclass_trait_name = "Function%d" % num_args
    superclass_trait = getattr(scala, superclass_trait_name)

   class ScalafiedFunction(superclass_trait):
      … % implementation of apply and __call__

   scalafied_function = ScalafiedFunction()

   java_abstract_cl_name = "%s$class" % superclass_n
   java_abstract_cl = getattr(scala, java_abstrac        ame)

   method = getattr(java_abstract_cl, "andThen")

   logic = lambda *args: method.__call__(scalafied_function,
                                         *args)

   setattr(scalafied_function, "andThen", logic)

   return scalafied_function
```

… use Python-style introspection (**`getattr`**) to obtain a reference to the concrete implementation of **`andThen`**, use a lambda expression to capture logic that invokes that concrete implementation (via **`__call__`**), and use **`setattr`** to add **`andThen`** to **`scalafied_function`**, binding it to the lambda expression.

Incidentally, the ability to use Python-style introspection on Java objects, in contrast to the more cumbersome Java reflection, is one of Jython's most useful features.



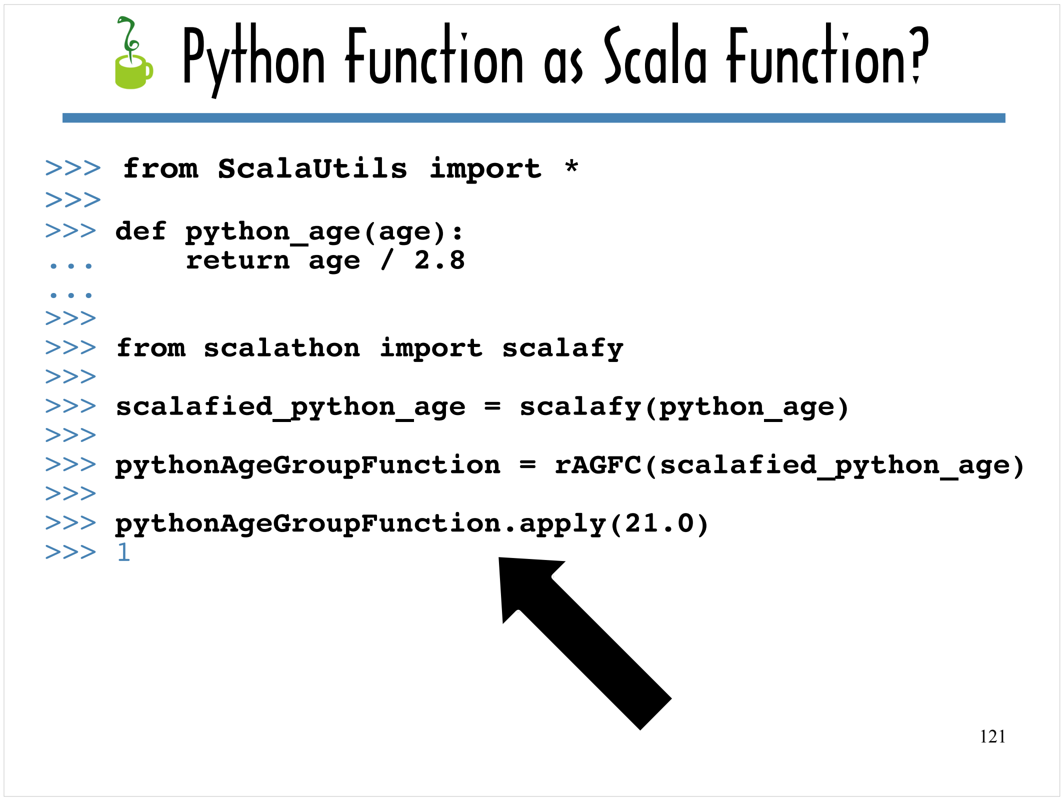
## Scalafying a Python Function

```
def scalafy(func):

    num_args = len(inspect.getargspec(func)[0])
    superclass_trait_name = "Function%d" % num_args
    superclass_trait = getattr(scala, superclass_trait_name)

   class ScalafiedFunction(superclass_trait):
      … % implementation of apply and __call__

   scalafied_function = ScalafiedFunction()

   java_abstract_cl_name = "%s$class" % superclass_n
   java_abstract_cl = getattr(scala, java_abstract        ame)

   method = getattr(java_abstract_cl, "andThen")

   logic = lambda *args: method.__call__(scalafied_function,
                                          *args)

   setattr(scalafied_function, "andThen", logic)

   return scalafied_function
```

120

… use Python-style introspection (getattr) to obtain a reference to the concrete implementation of andThen, use a lambda expression to capture logic that Invokes that concrete implementation (via __call__), and use setattr to add AndThen to scalafied_function, binding it to the lambda expression.

Incidentally, the ability to use Python-style introspection on Java objects is one Of Jython's most useful features.



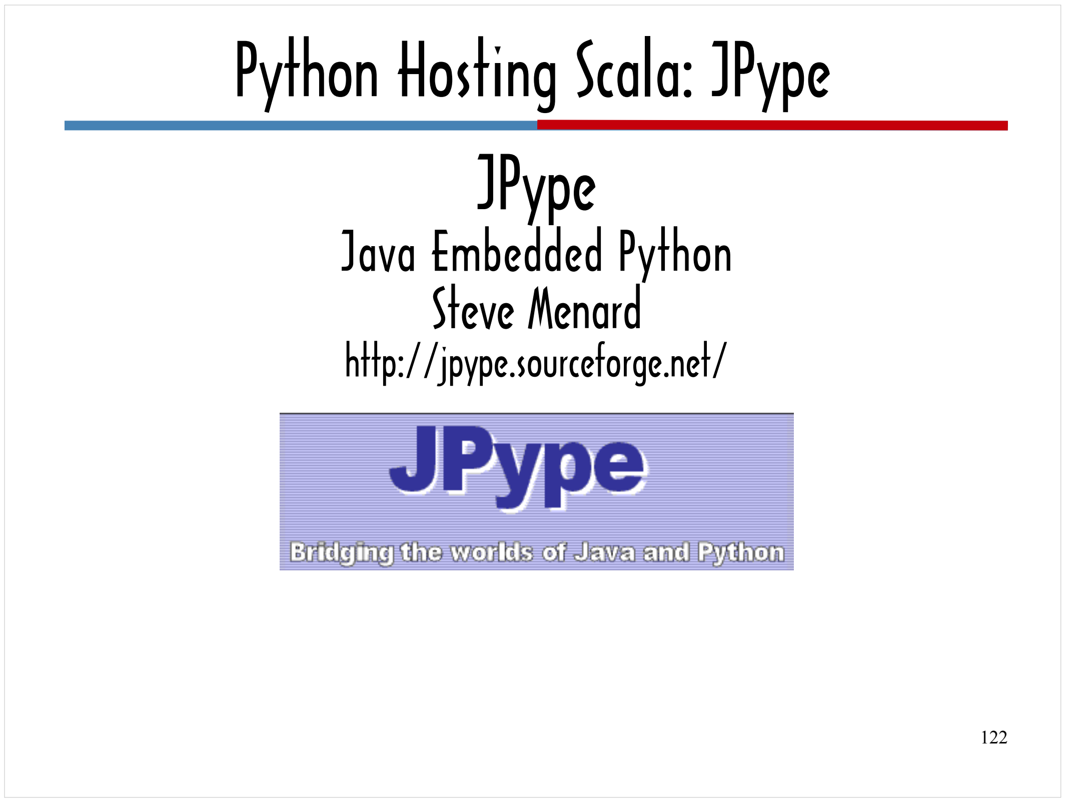
## Python Function as Scala Function?

```
>>> from ScalaUtils import *
>>>
>>> def python_age(age):
...     return age / 2.8
...
>>>
>>> from scalathon import scalafy
>>>
>>> scalafied_python_age = scalafy(python_age)
>>>
>>> pythonAgeGroupFunction = rAGFC(scalafied_python_age)
>>>
>>> pythonAgeGroupFunction.apply(21.0)
>>> 1
```

121

In this Jython console session, you can see that passing the scalafied version of **`python_age`** to **`reptileAgeGroupFunctionFactory`** (**`pythonAgeGroupFunction`**) now yields a function that behaves like a Scala function.

Given Python's age (21), **`pythonAgeGroupFunction`** returns the age group Python would be assigned to if Python were a snake instead of a programming language.



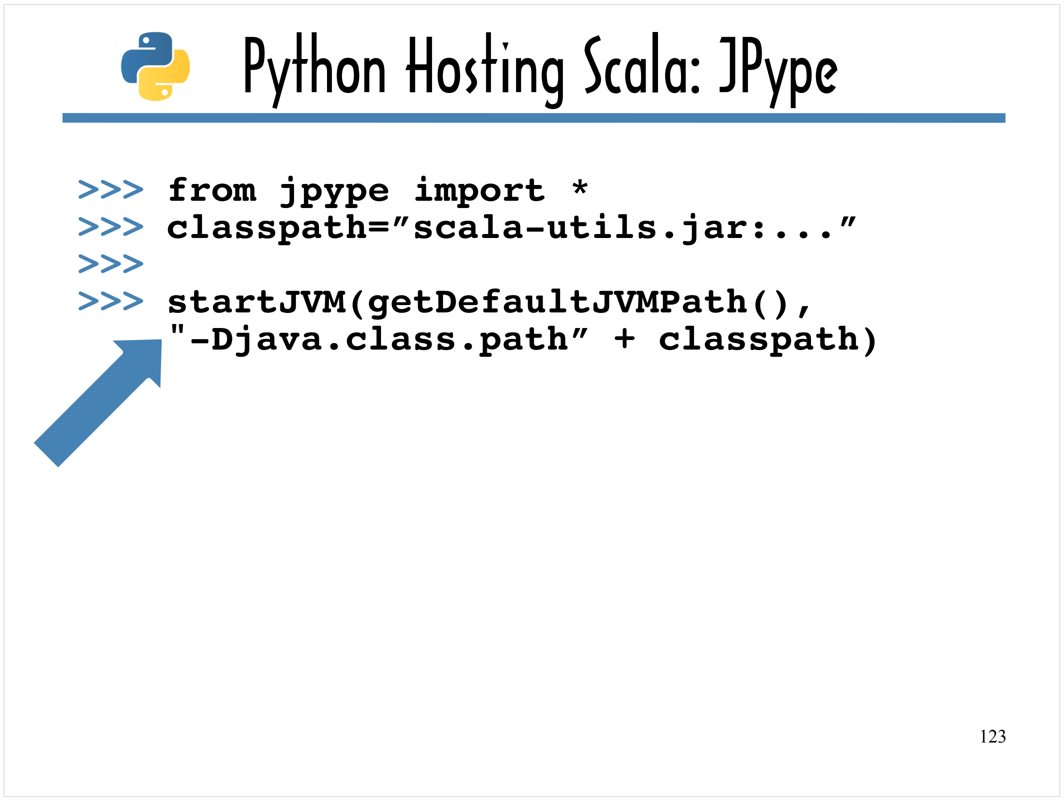
## Python Hosting Scala: JPype

JPype
Java Embedded Python
Steve Menard
http://jpype.sourceforge.net/

JPype
Bridging the worlds of Java and Python

JPype is the first of 3 tools that enable you to invoke methods and functions defined in Scala from Python (ie the C-based implementation of Python) that I'm going to cover.

JPype leverages the Python/C interface (via C++) and JNI.



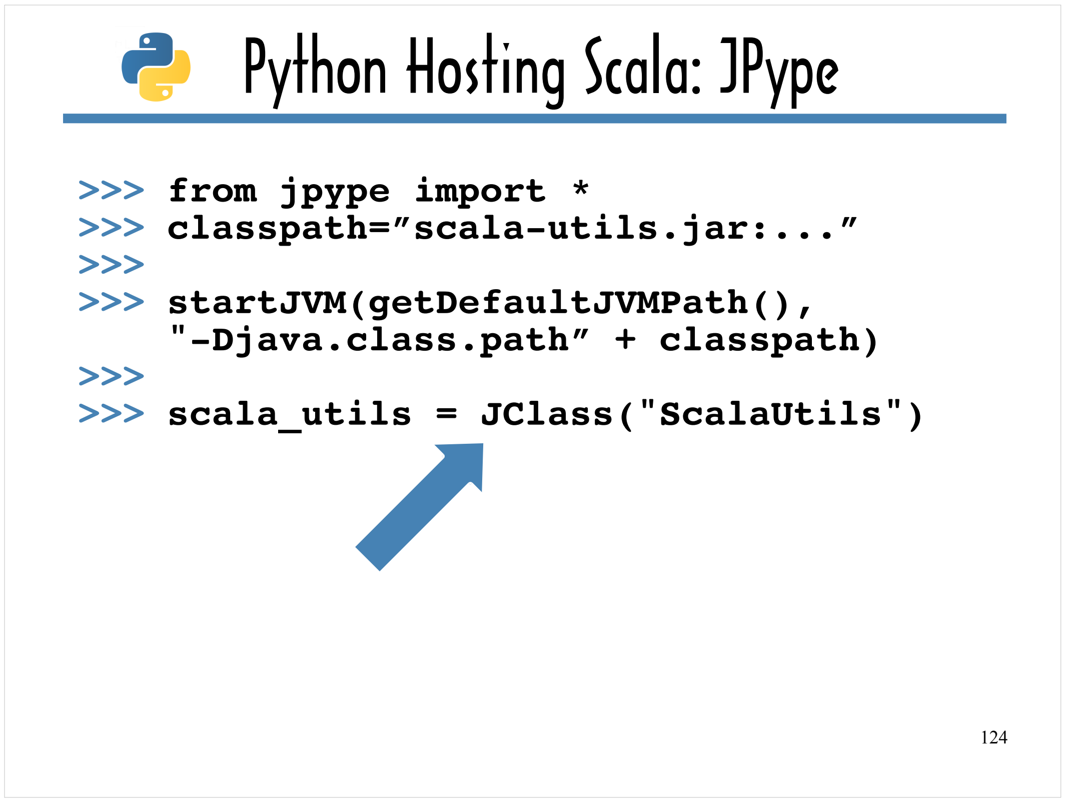
## Python Hosting Scala: JPype

```
>>> from jpype import *
>>> classpath="scala-utils.jar:..."
>>>
>>> startJVM(getDefaultJVMPath(),
    "-Djava.class.path" + classpath)
```

Every JPype console session (or Python script) begins with embedding a JVM by calling **`startJVM`**, as shown here. Downstream, **`startJVM`** calls the JNI function **`JNI_CreateJavaVM`**.

If you're not using JNI from within a Java program, you need to spin up a JVM using **`JNI_CreateJavaVM`**, before you can make any other JNI calls.

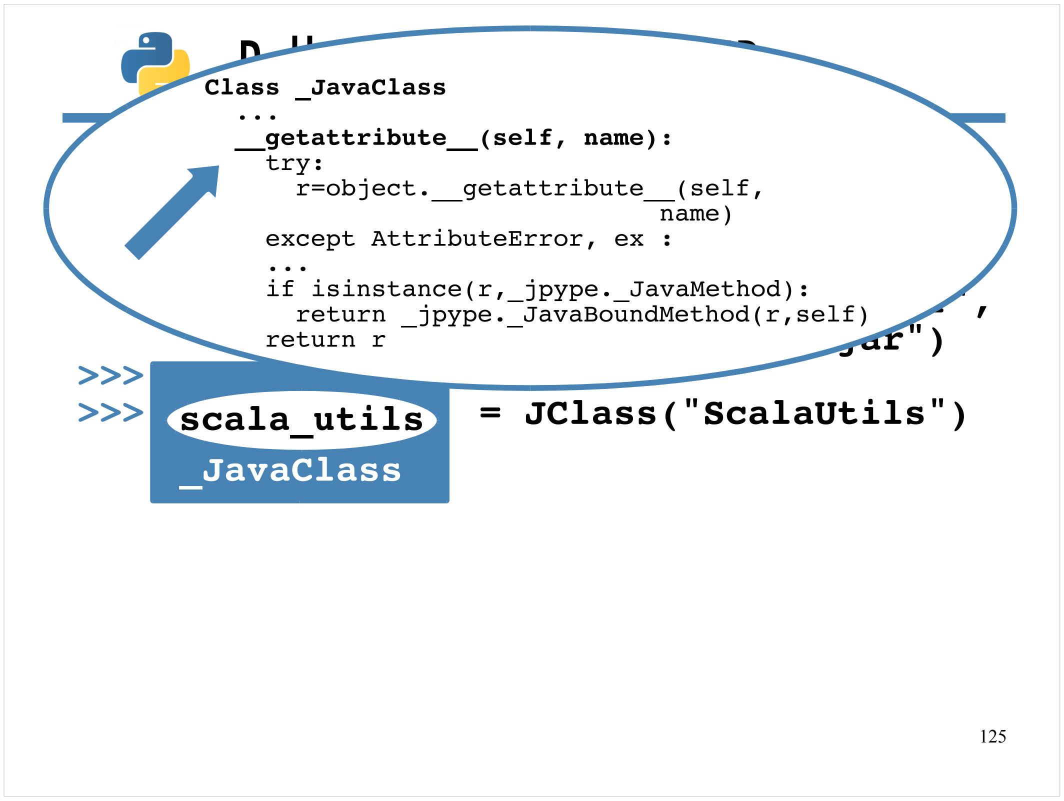
The ellipses in the classpath, represent the path to **`scala-library.jar`**, which is included in the standard Scala distribution.



## Python Hosting Scala: JPype

```
>>> from jpype import *
>>> classpath=”scala-utils.jar:...”
>>>
>>> startJVM(getDefaultJVMPath(),
    "-Djava.class.path” + classpath)
>>>
>>> scala_utils = JClass("ScalaUtils")
```

You can import a Scala object using JPype's **`JClass`** function, which ..

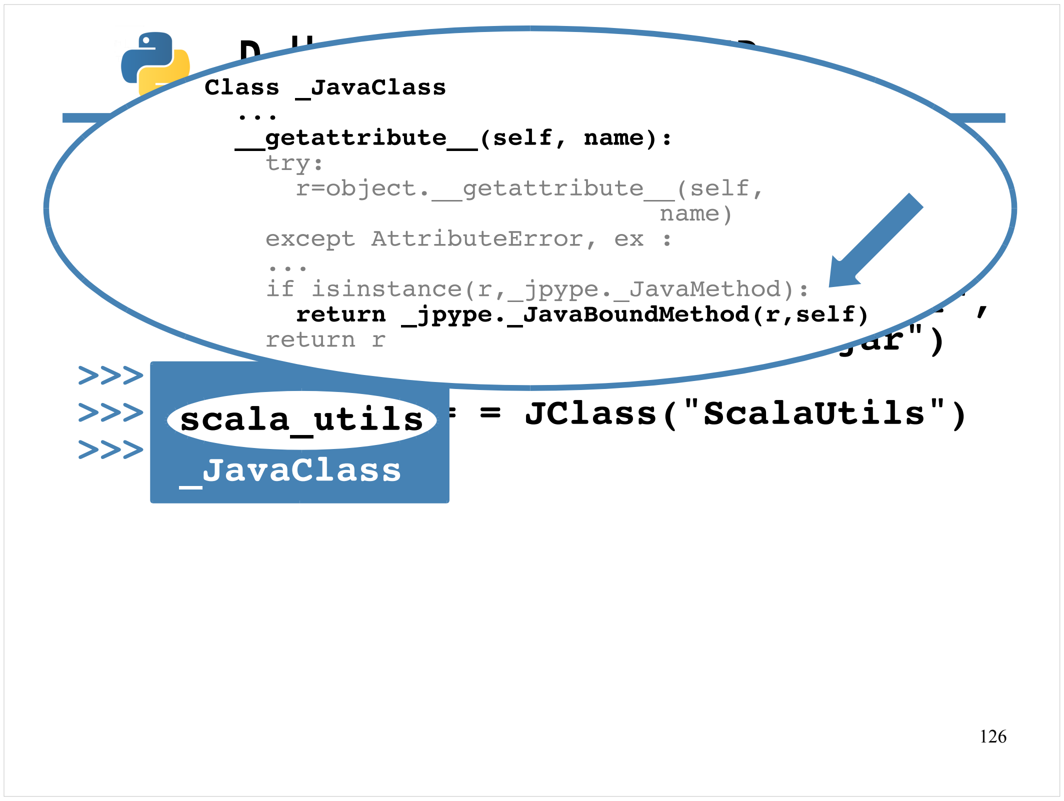


```
Class _JavaClass
  ...
  __getattribute__(self, name):
    try:
      r=object.__getattribute__(self,
                                name)
    except AttributeError, ex :
    ...
    if isinstance(r,_jpype._JavaMethod):
      return _jpype._JavaBoundMethod(r,self)
    return r
```

```
>>>
>>> scala_utils = JClass("ScalaUtils")
```

returns an instance of _JavaClass, a Python class that wraps a Java (or Scala) class to provide easy access to its methods and attributes.

In the _JavaClass definition, **__getattribute__** ...



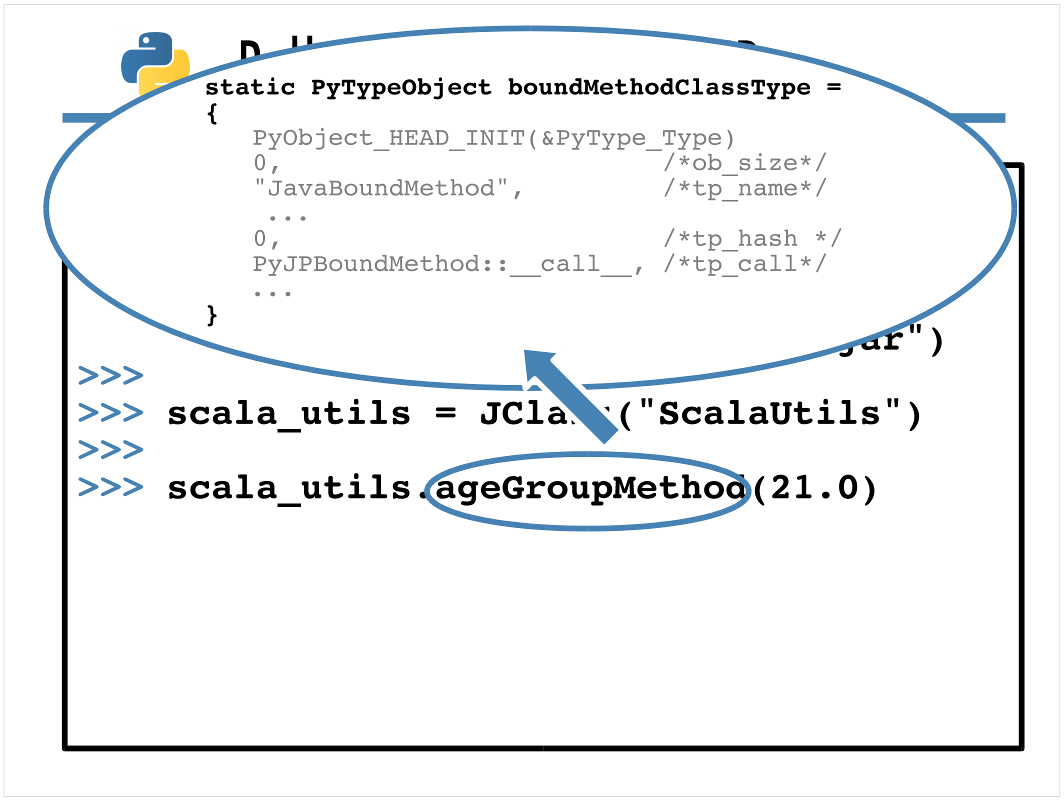
```
Class _JavaClass
  ...
  __getattribute__(self, name):
    try:
      r=object.__getattribute__(self,
                                name)
    except AttributeError, ex :
    ...
    if isinstance(r,_jpype._JavaMethod):
      return _jpype._JavaBoundMethod(r,self)
    return r
```

```
>>>
>>> scala_utils = JClass("ScalaUtils")
>>>
```

...is implemented to use JNI to invoke a Java function on the Java class that the _JavaClass instance wraps when dot notation is used to invoke a method on an instance of _JavaClass. All of this handling works for Scala as well.

If an AttributeError is thrown, meaning there is not a Python-based definition for the method called on the _JavaClass instance, JPype assumes the method is defined in Java.

The line highlighted with an arrow shows the first step in accessing a method defined in Java. It passes a reference to the Java method to **`_JavaBoundMethod`**, which ...



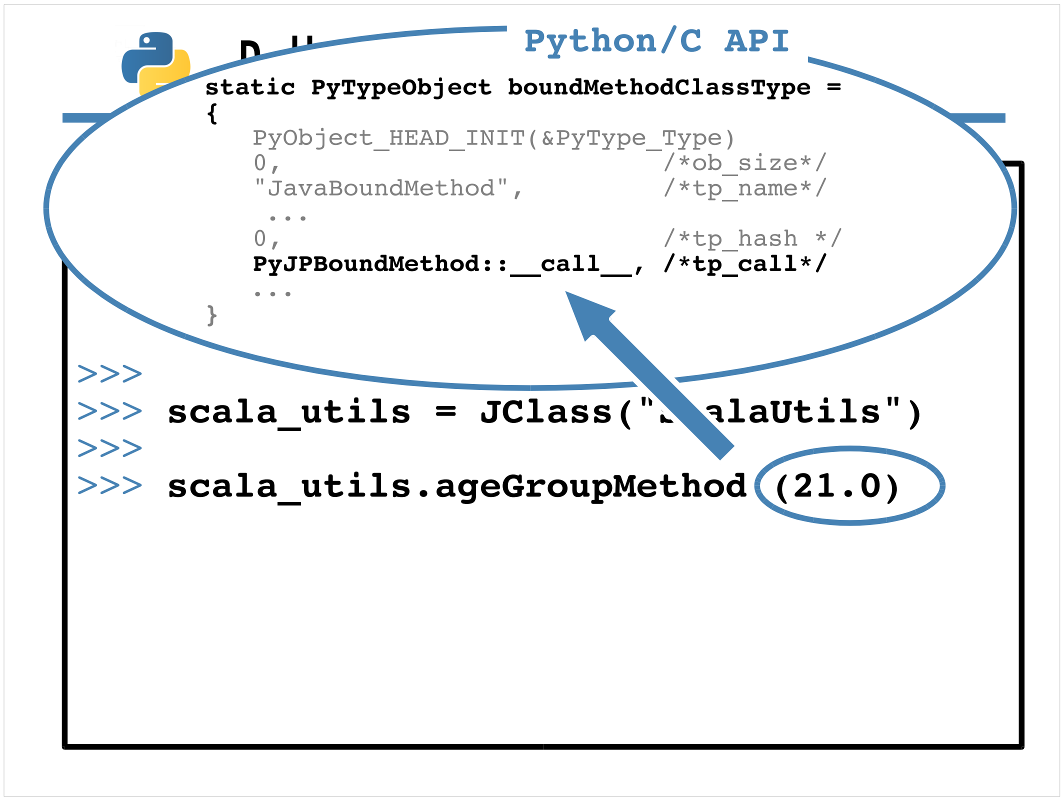
```
static PyTypeObject boundMethodClassType =
{
   PyObject_HEAD_INIT(&PyType_Type)
   0,                           /*ob_size*/
   "JavaBoundMethod",           /*tp_name*/
    ...
   0,                           /*tp_hash */
   PyJPBoundMethod::__call__, /*tp_call*/
   ...
}
```

```
>>>
>>> scala_utils = JClass("ScalaUtils")
>>>
>>> scala_utils.ageGroupMethod(21.0)
```

… creates a Jpype boundMethodClassType object.

JPype defines the boundMethodClassType using the Python/C API's PyTypeObject structure for defining custom types.

As indicated by the comments following each field in the inset showing an excerpt from the boundMethodClassType definition, when you use the PyTypeObject structure you can specify property values for the custom type, as well as customized versions of methods standard Python objects support ...



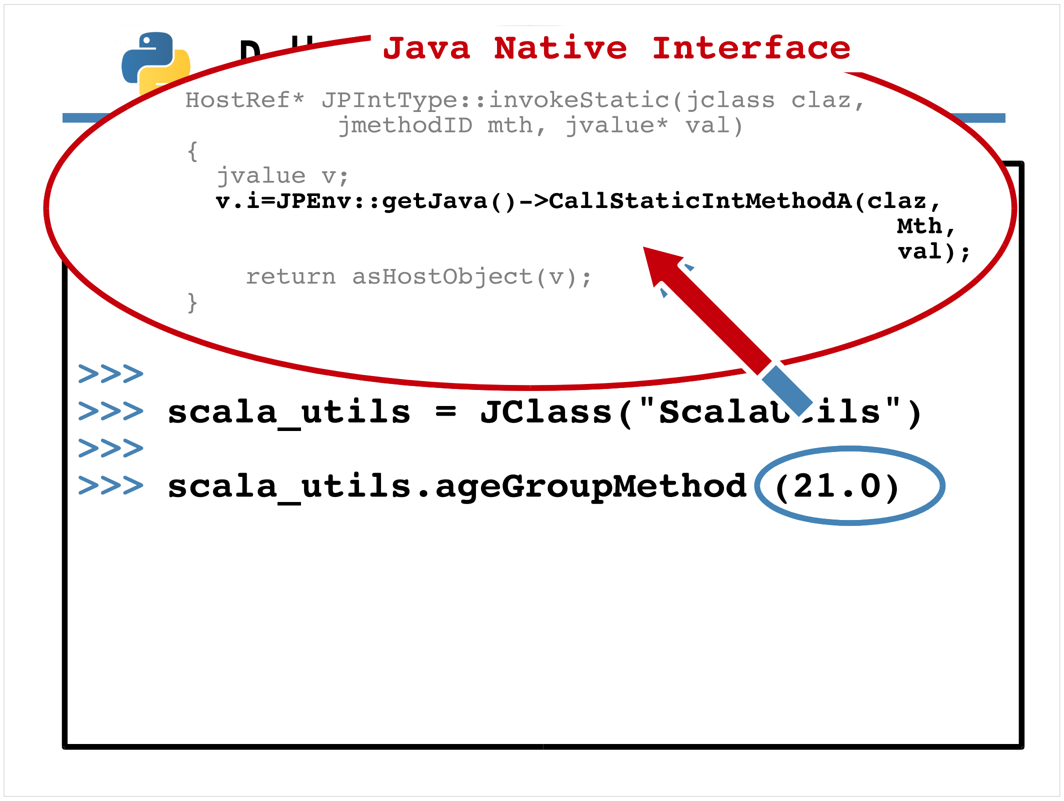
```
static PyTypeObject boundMethodClassType =
{
   PyObject_HEAD_INIT(&PyType_Type)
   0,                           /*ob_size*/
   "JavaBoundMethod",           /*tp_name*/
    ...
   0,                           /*tp_hash */
   PyJPBoundMethod::__call__,   /*tp_call*/
   ...
}
```

```
>>>
>>> scala_utils = JClass("ScalaUtils")
>>>
>>> scala_utils.ageGroupMethod (21.0)
```

...like **`__call__`**.

The method tagged with “tp_call” is invoked when an argument list (in this case, “(21.0)”) is tacked onto the method identifier (in this case, “ageGroupMethod”).

The **`__call__`** implementation, uses JNI to determine the number and type(s) of argument(s) and based the number and type(s) of argument(s), makes the corresponding JNI invocation call, which in this case is ...

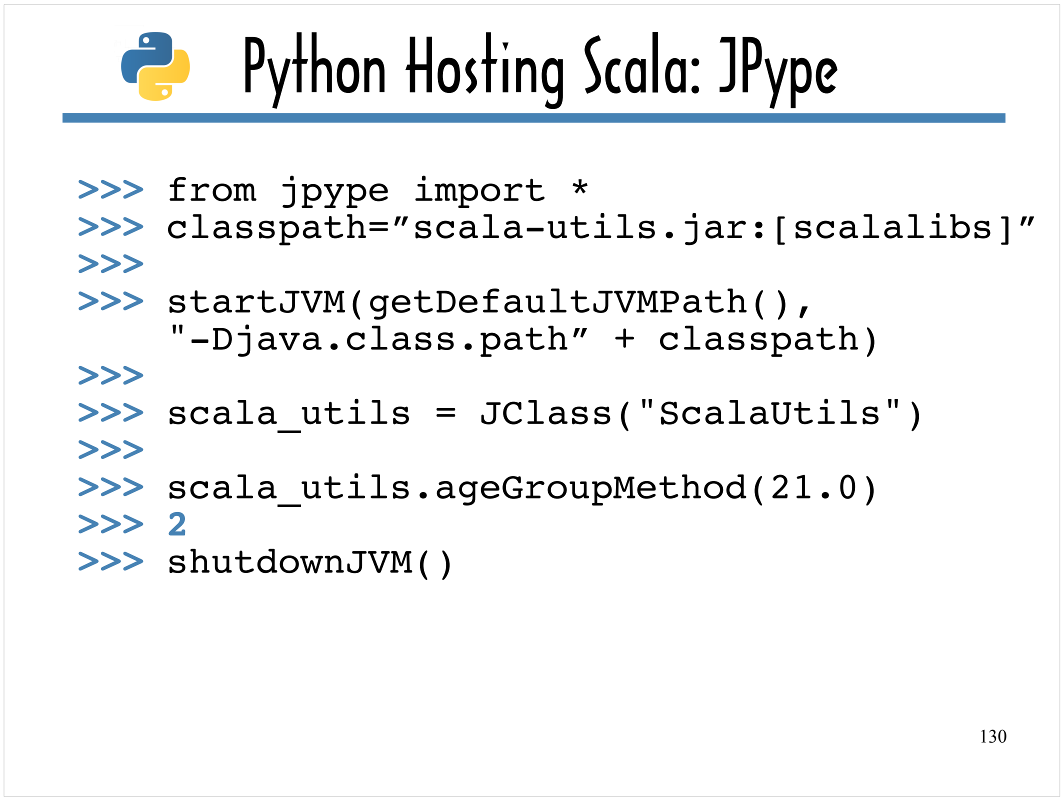


Java Native Interface

```
HostRef* JPIntType::invokeStatic(jclass claz,
          jmethodID mth, jvalue* val)
{
  jvalue v;
  v.i=JPEnv::getJava()->CallStaticIntMethodA(claz,
                                              Mth,
                                              val);

    return asHostObject(v);
}
```

```
>>>
>>> scala_utils = JClass("ScalaUtils")
>>>
>>> scala_utils.ageGroupMethod (21.0)
```

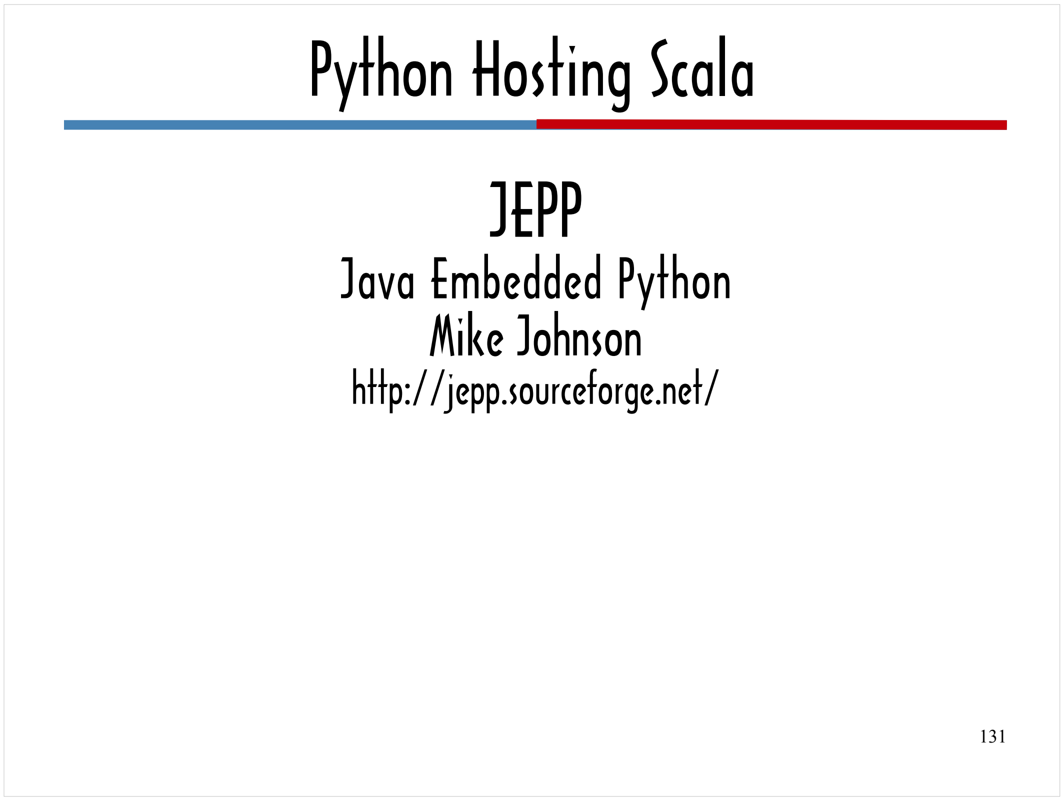
**`CallStaticIntMethodA`**. The “A” stands for “arguments”. JNI also provides **`CallStaticIntMethod`** for methods that return an Int, but don't take any arguments.



## Python Hosting Scala: JPype

```
>>> from jpype import *
>>> classpath=”scala-utils.jar:[scalalibs]”
>>>
>>> startJVM(getDefaultJVMPath(),
    "-Djava.class.path” + classpath)
>>>
>>> scala_utils = JClass("ScalaUtils")
>>>
>>> scala_utils.ageGroupMethod(21.0)
>>> 2
>>> shutdownJVM()
```

Here's the complete console session, including the return value from the **`ScalaUtils#ageGroupMethod`** call, **`2`**, and the **`shutdownJVM`** call that should conclude any console session or script that beging with **`startJVM`**.


## Python Hosting Scala

JEPP

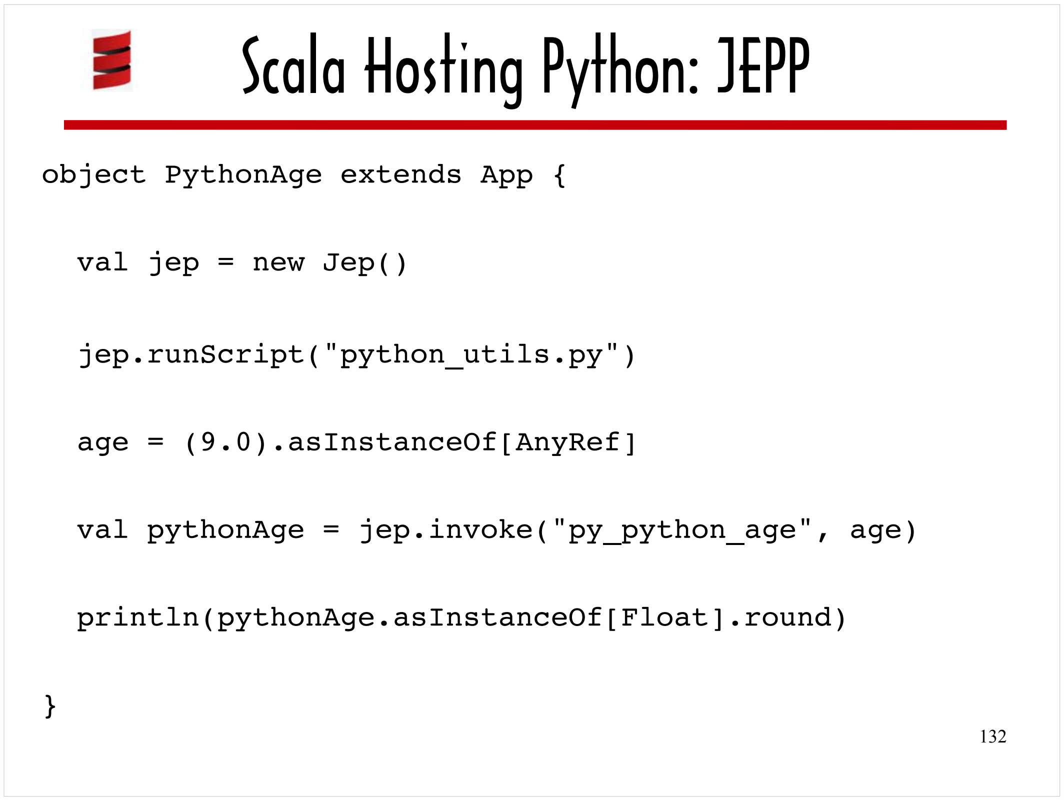
Java Embedded Python

Mike Johnson

http://jepp.sourceforge.net/

JEPP is another tool that enables you to write Python scripts that access Scala code, and like JPype, it uses both JNI and the Python\C API.

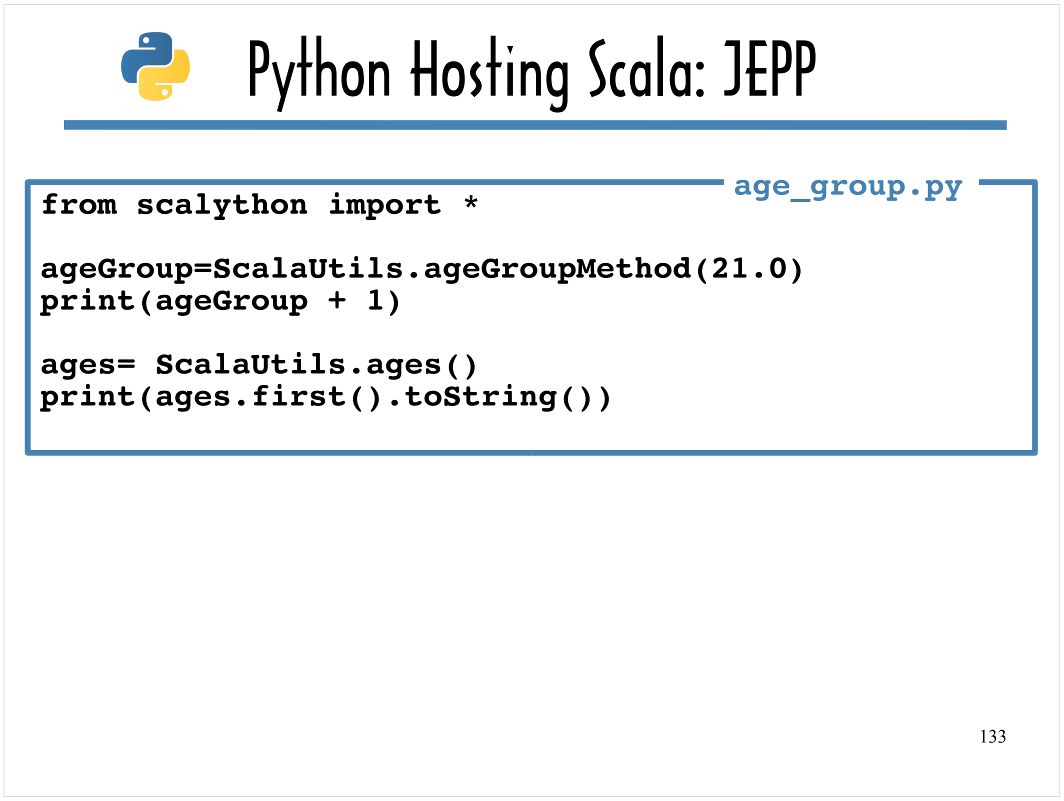
Towards the beginning of this talk I showed you how JEPP could be used to access a Python script from within a Scala program.



## Scala Hosting Python: JEPP

```
object PythonAge extends App {

  val jep = new Jep()

  jep.runScript("python_utils.py")

  age = (9.0).asInstanceOf[AnyRef]

  val pythonAge = jep.invoke("py_python_age", age)

  println(pythonAge.asInstanceOf[Float].round)

}
```

For reference, here's that Scala example code that invokes **`py_python_age`**, which is defined in a Python script.



## Python Hosting Scala: JEPP

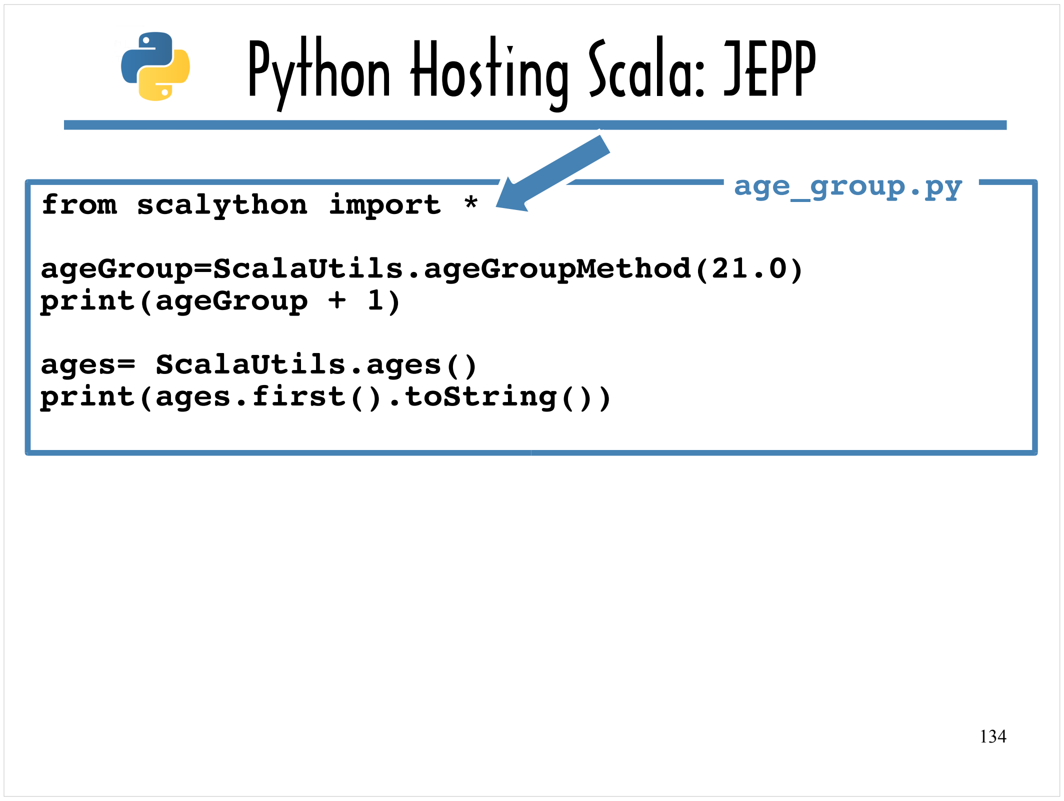
age_group.py

```
from scalython import *

ageGroup=ScalaUtils.ageGroupMethod(21.0)
print(ageGroup + 1)

ages= ScalaUtils.ages()
print(ages.first().toString())
```

Here's **age_group.py**, a Python script that uses JEPP to access the **ageGroupMethod**, which is defined in ScalaUtils in and written in Scala.

The reason for printing out **ageGroup + 1**, instead of just **ageGroup** is to show that Python recognizes that the return value of the **ScalaUtils#ageGroupMethod** call is an integer. As you will see, the number **3**, prints out when you run this script.

The reason for printing **ages_first().toString()** instead of just **ages** is to show that you can easily call Scala-based methods, like, **ListMap#first** on the **ages** ListMap from within a Python script.



## Python Hosting Scala: JEPP

age_group.py

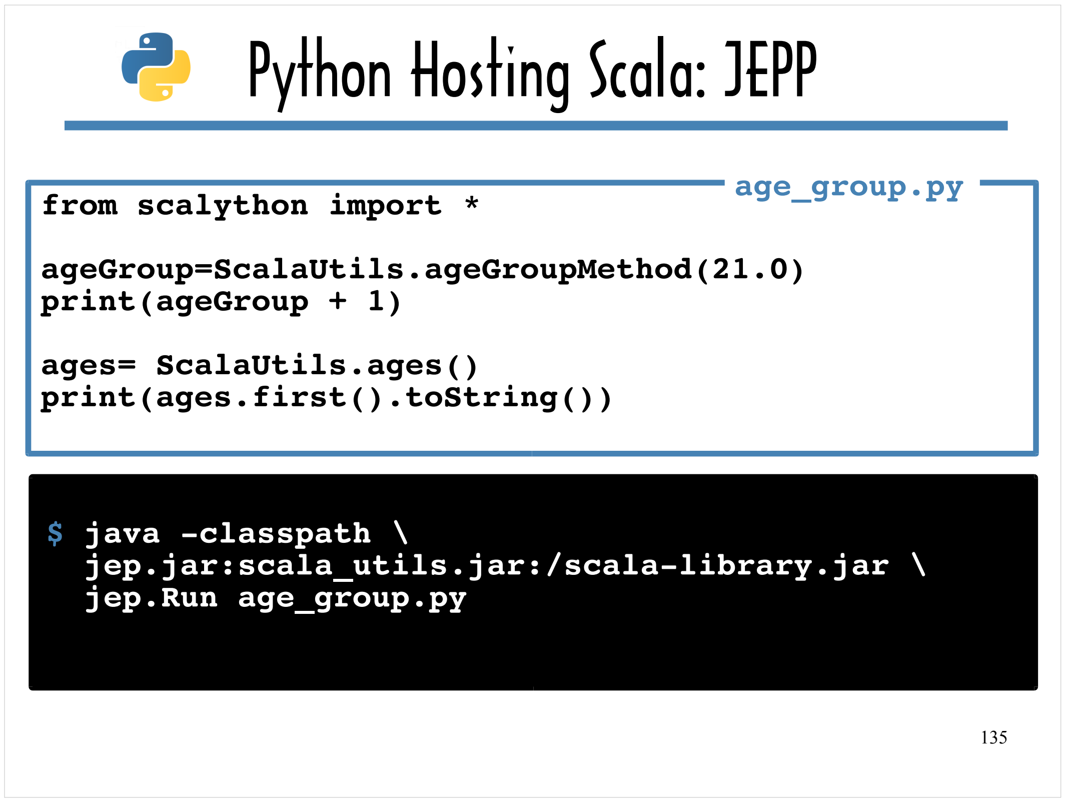
```
from scalython import *

ageGroup=ScalaUtils.ageGroupMethod(21.0)
print(ageGroup + 1)

ages= ScalaUtils.ages()
print(ages.first().toString())
```

In the previous examples that access the ScalaUtils micro-library, I have not needed to put ScalaUtils in a package.

But in order to access ScalaUtils using JEPP, I did need to modify **`ScalaUtils.scala`** by adding the following statement to specify a package for ScalaUtils: **`package scalython`**.

The name “scalython” yet another one of the permutations of “scala” and “python” I came up with.

Once imported, I could call methods on ScalaUtils, like **`ScalaUtils#ageGroupMethod`**, and access properties on ScalaUtils, from within the Python script, using dot notation.



## Python Hosting Scala: JEPP

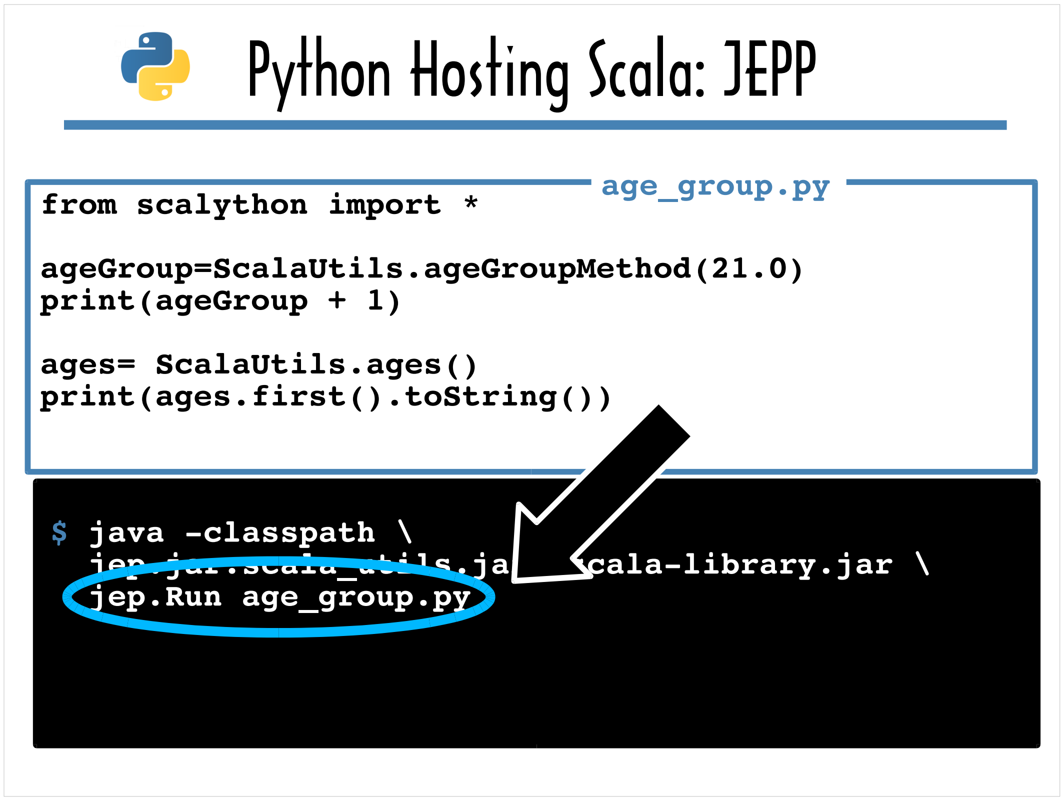
age_group.py

```
from scalython import *

ageGroup=ScalaUtils.ageGroupMethod(21.0)
print(ageGroup + 1)

ages= ScalaUtils.ages()
print(ages.first().toString())
```

```
$ java -classpath \
  jep.jar:scala_utils.jar:/scala-library.jar \
  jep.Run age_group.py
```

In this slide, I am including the command that runs this Python script to highlight the biggest difference between JEPP and JPype: JEPP does not embed a JVM into the Python script. Recall that in order to use JNI, you either need to embed a JVM or make JNI API calls from within a Java program.

Running the **`age_group.py`** JEPP script requires the **`java`** command line interpreter. You need to pass the name of the Python script …



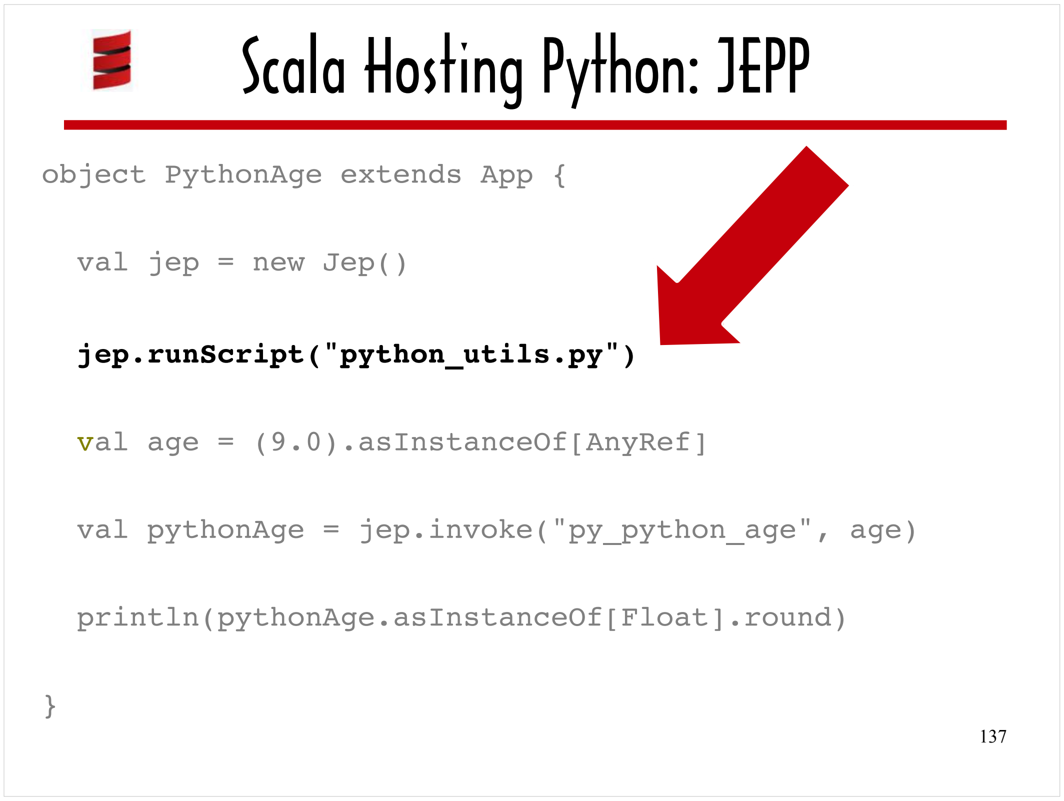
## Python Hosting Scala: JEPP

age_group.py

```
from scalython import *

ageGroup=ScalaUtils.ageGroupMethod(21.0)
print(ageGroup + 1)

ages= ScalaUtils.ages()
print(ages.first().toString())
```

```
$ java -classpath \
  jep.jar:scala_utils.jar:scala-library.jar \
  jep.Run age_group.py
```

to jep.Run, which is a Java program packaged with the JEPP distribution and that calls the same **`Jep#runScript`** method...

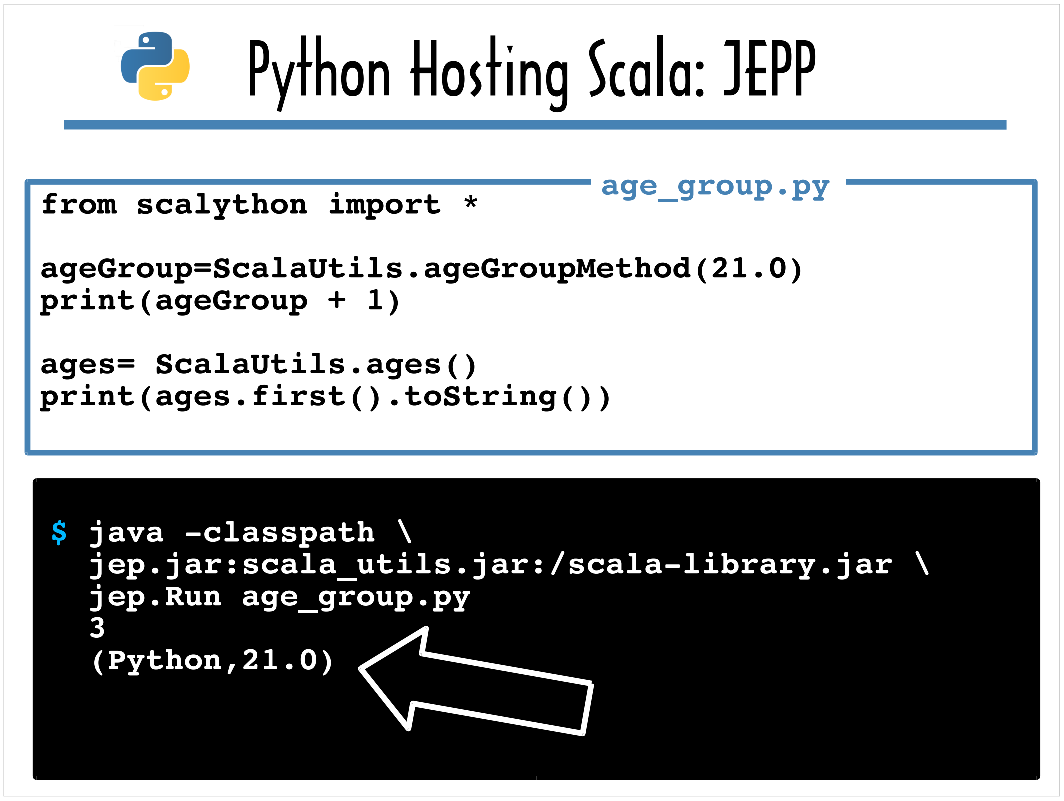


## Scala Hosting Python: JEPP

```
object PythonAge extends App {

  val jep = new Jep()

  jep.runScript("python_utils.py")

  val age = (9.0).asInstanceOf[AnyRef]

  val pythonAge = jep.invoke("py_python_age", age)

  println(pythonAge.asInstanceOf[Float].round)

}
```

… I used to process **`python_utils.py`** in the example I used to show how to access the Python micro-library from within a Scala program.

The jep.Run program does more than just wrap a call to **`Jep#runScript,`** though. It supports an interactive option for running the JEPP console and a swingApp option for scripting Swing applications.



## Python Hosting Scala: JEPP

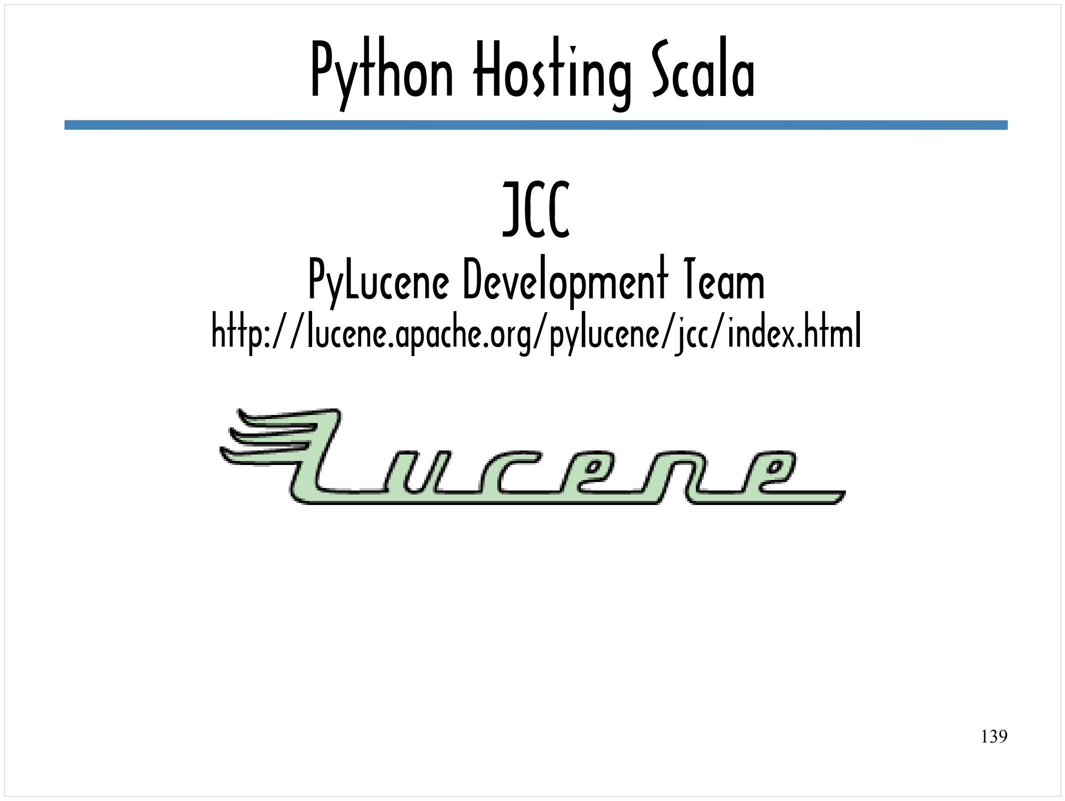
age_group.py

```
from scalython import *

ageGroup=ScalaUtils.ageGroupMethod(21.0)
print(ageGroup + 1)

ages= ScalaUtils.ages()
print(ages.first().toString())
```

```
$ java -classpath \
  jep.jar:scala_utils.jar:/scala-library.jar \
  jep.Run age_group.py
  3
  (Python,21.0)
```

This slide shows the output when **age_group.py** is passed to jep.Run.

If you need to call Scala from within a Python script, JEPP enables you to do all of your coding in Python – but your program ultimately would need to be packaged as a Java program.



## Python Hosting Scala

JCC

PyLucene Development Team

http://lucene.apache.org/pylucene/jcc/index.html

139

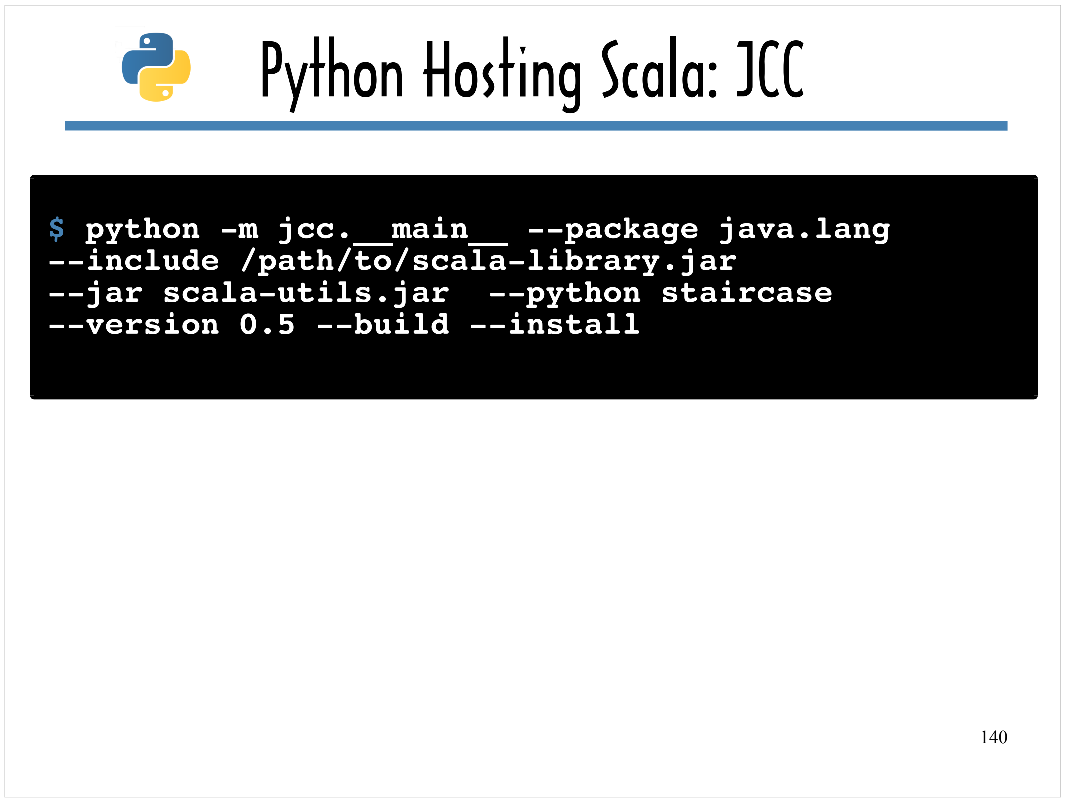
The last tool that facilitates hosting Scala from Python that I'm going to cover is JCC.

It was developed by the PyLucene team to enable Python developers to use the Java-based search tool, Lucene.

With a single command on the command line, JCC:
1. generates C++ code that exposes a specified Java library via JNI
2. generates a Python\C++ extension that wraps the JNI code
3. generates and installs a Python package that provides access to the specified Java library.

The next slide shows the single JCC command I used to generate a Python package that wraps the **`ScalaUtils#age_group_method`** from my Scala micro-library.

Although JCC works well with **`ScalaUtils#age_group_method`**, it's not likely to work out-of-the-box for a real world Scala/Python integration project. The micro-library does not include Scala features that diverge very much from standard Java. I ran into code generation issues when I tried to use JCC to generate a Python package that makes all the Scala core classes accessible from Python. I included JCC in the slide deck because I thought its end-to-end generation approach would interest polyglot programming enthusiasts.

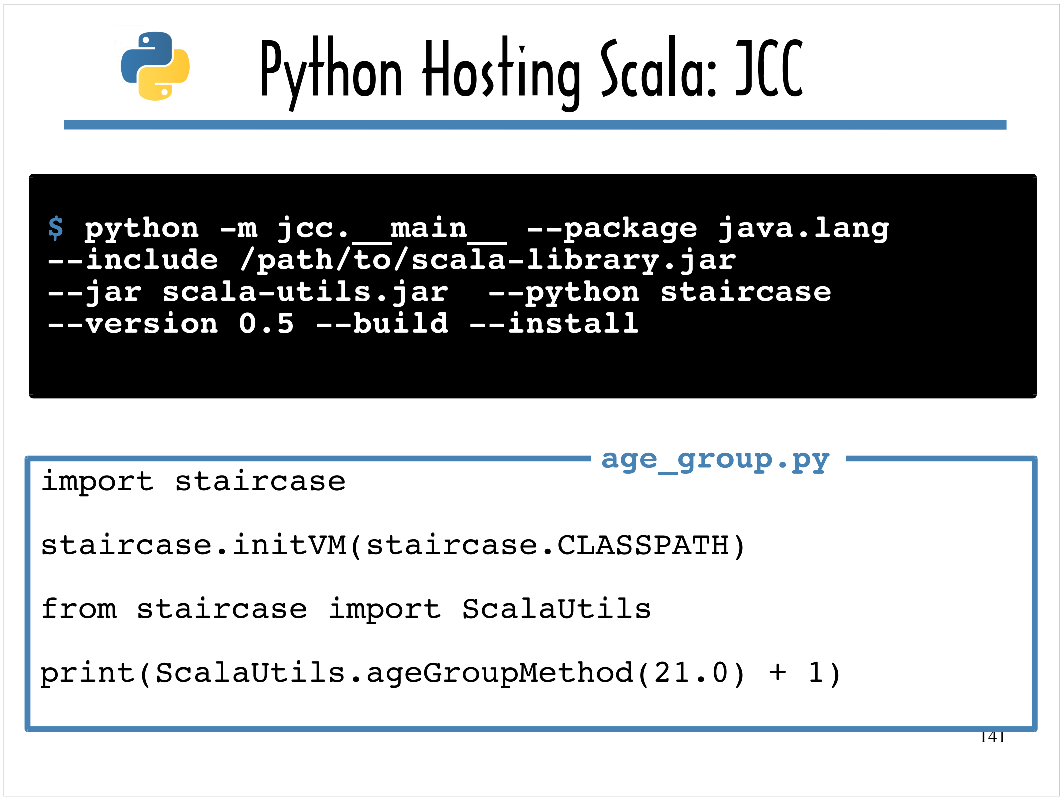


## Python Hosting Scala: JCC

```
$ python -m jcc.__main__ --package java.lang
--include /path/to/scala-library.jar
--jar scala-utils.jar  --python staircase
--version 0.5 --build --install
```

Here's the command I used to generate a Python package called “staircase” that wraps my Scala micro-library, which I archived in a Java .jar file called **`scala-utils.jar`**.

The **`--jar`** option is used to specify the Java library JCC should wrap. The **`--python`** option and the **`--version`** option specify the name and version of the Python package JCC should generate. The **`--build`** and **--install** options indicate that the Python package should be installed. The **`--include`** option is used to specify any files that should be included in the Java CLASSPATH for the wrapped Java library.



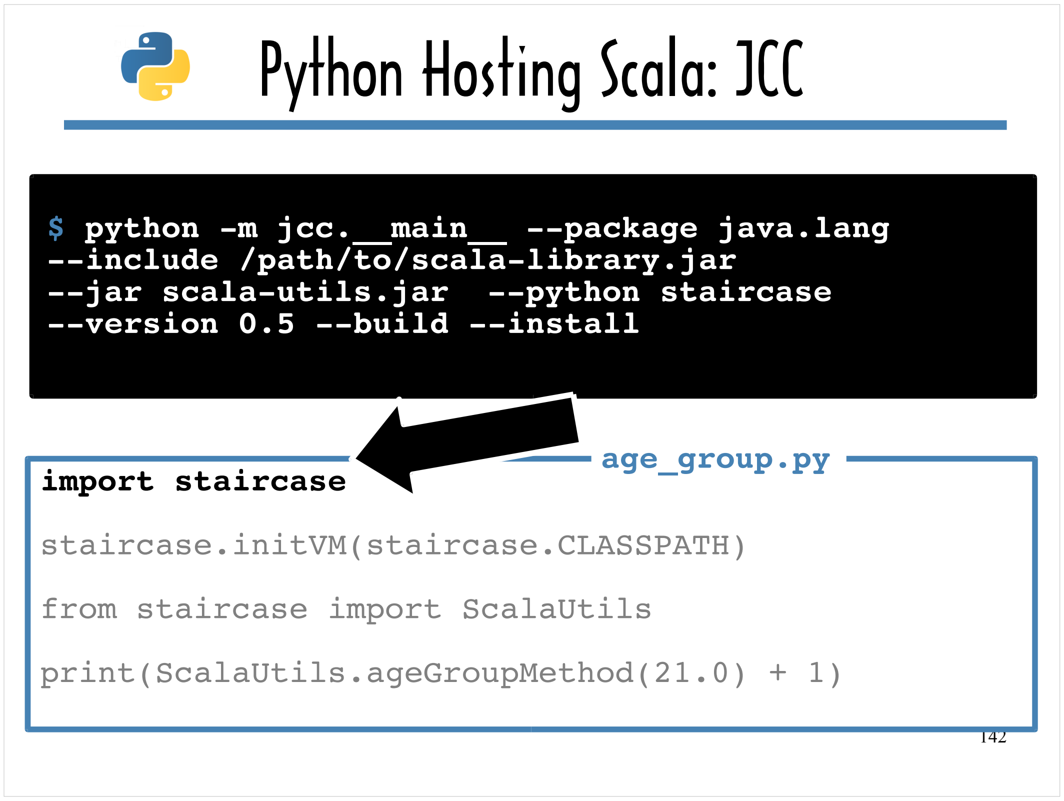
## Python Hosting Scala: JCC

```
$ python -m jcc.__main__ --package java.lang
--include /path/to/scala-library.jar
--jar scala-utils.jar  --python staircase
--version 0.5 --build --install
```

age_group.py

```
import staircase

staircase.initVM(staircase.CLASSPATH)

from staircase import ScalaUtils

print(ScalaUtils.ageGroupMethod(21.0) + 1)
```

Using the generated Python package is easy, as you can see from this script.



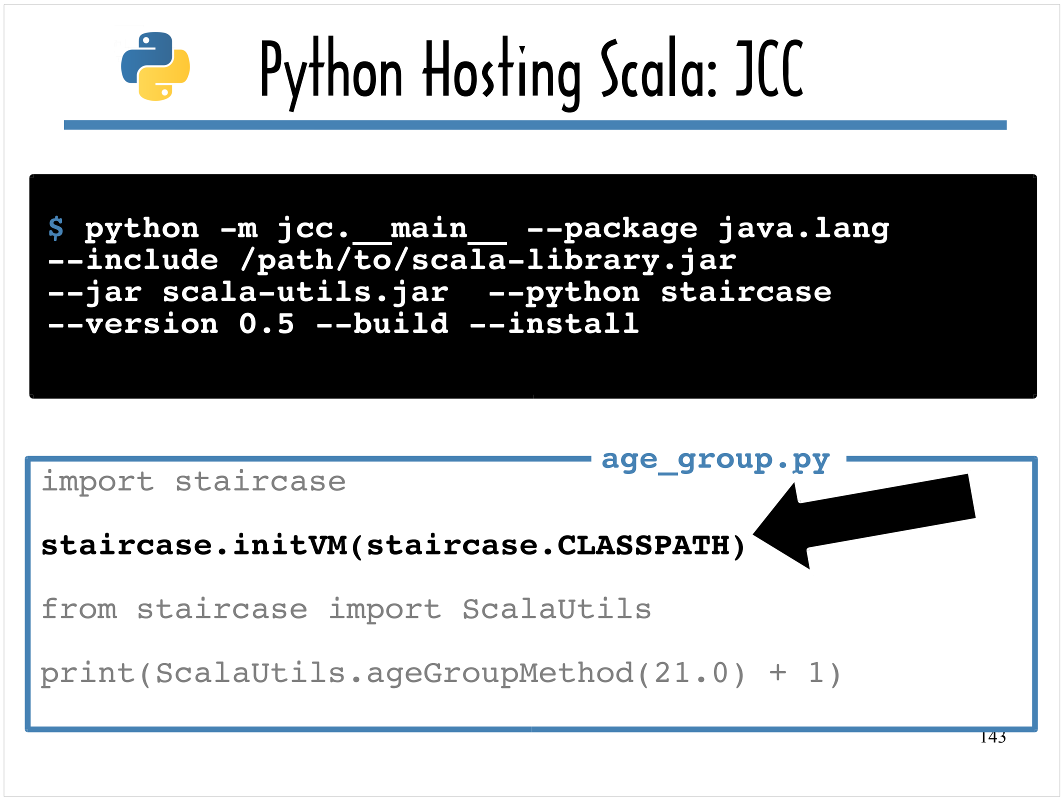
## Python Hosting Scala: JCC

```
$ python -m jcc.__main__ --package java.lang
--include /path/to/scala-library.jar
--jar scala-utils.jar  --python staircase
--version 0.5 --build --install
```

age_group.py

```
import staircase

staircase.initVM(staircase.CLASSPATH)

from staircase import ScalaUtils

print(ScalaUtils.ageGroupMethod(21.0) + 1)
```

First, you can import the generated package, the way you would typically import a Python module.



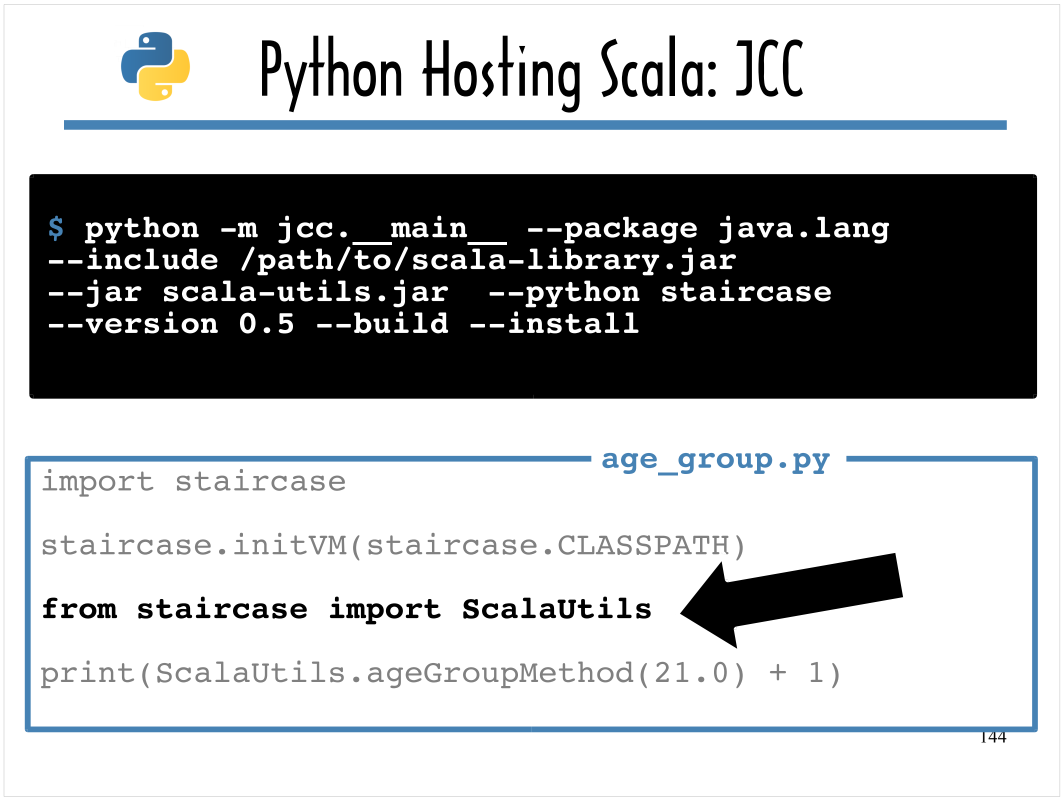
## Python Hosting Scala: JCC

```
$ python -m jcc.__main__ --package java.lang
--include /path/to/scala-library.jar
--jar scala-utils.jar  --python staircase
--version 0.5 --build --install
```

age_group.py

```
import staircase

staircase.initVM(staircase.CLASSPATH)

from staircase import ScalaUtils

print(ScalaUtils.ageGroupMethod(21.0) + 1)
```

Next, you need to call **`initVM`**, passing it the generated **`CLASSPATH`**, as shown here, substituting the name of your package for **`staircase`**.



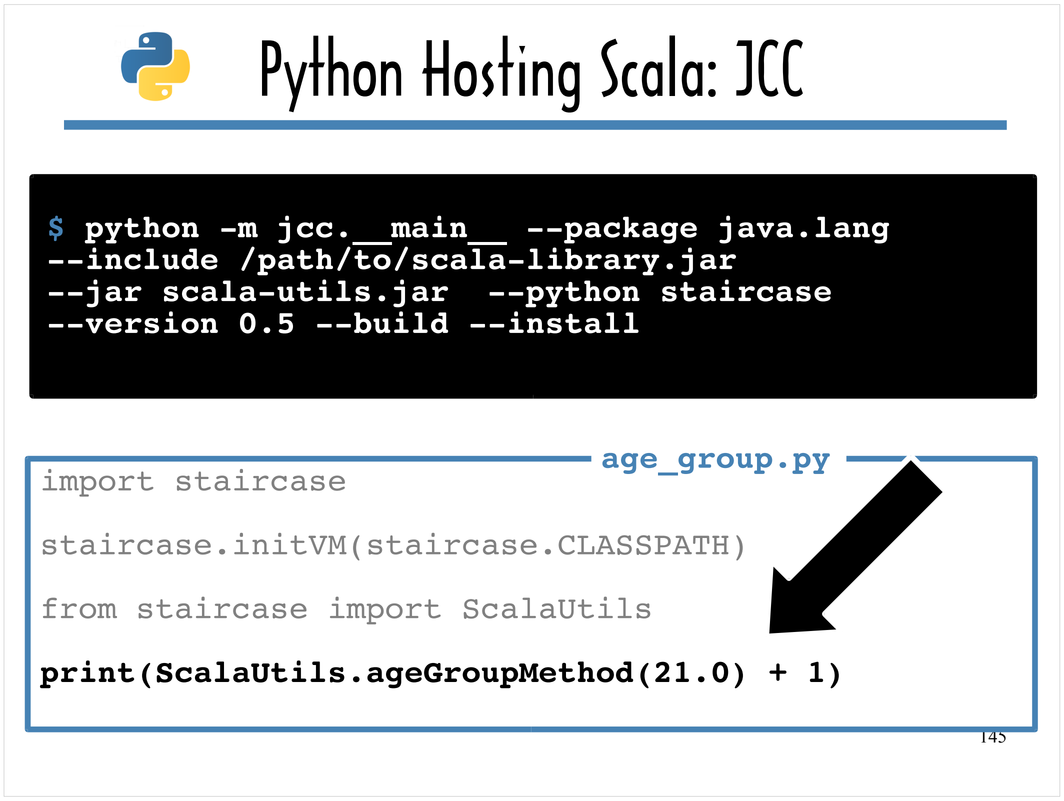
## Python Hosting Scala: JCC

```
$ python -m jcc.__main__ --package java.lang
--include /path/to/scala-library.jar
--jar scala-utils.jar  --python staircase
--version 0.5 --build --install
```

**age_group.py**

```
import staircase

staircase.initVM(staircase.CLASSPATH)

from staircase import ScalaUtils

print(ScalaUtils.ageGroupMethod(21.0) + 1)
```

Now that the JVM knows about everything on the **CLASSPATH** for the specified Scala library, you can import your Scala library.



## Python Hosting Scala: JCC

```
$ python -m jcc.__main__ --package java.lang
--include /path/to/scala-library.jar
--jar scala-utils.jar  --python staircase
--version 0.5 --build --install
```

age_group.py

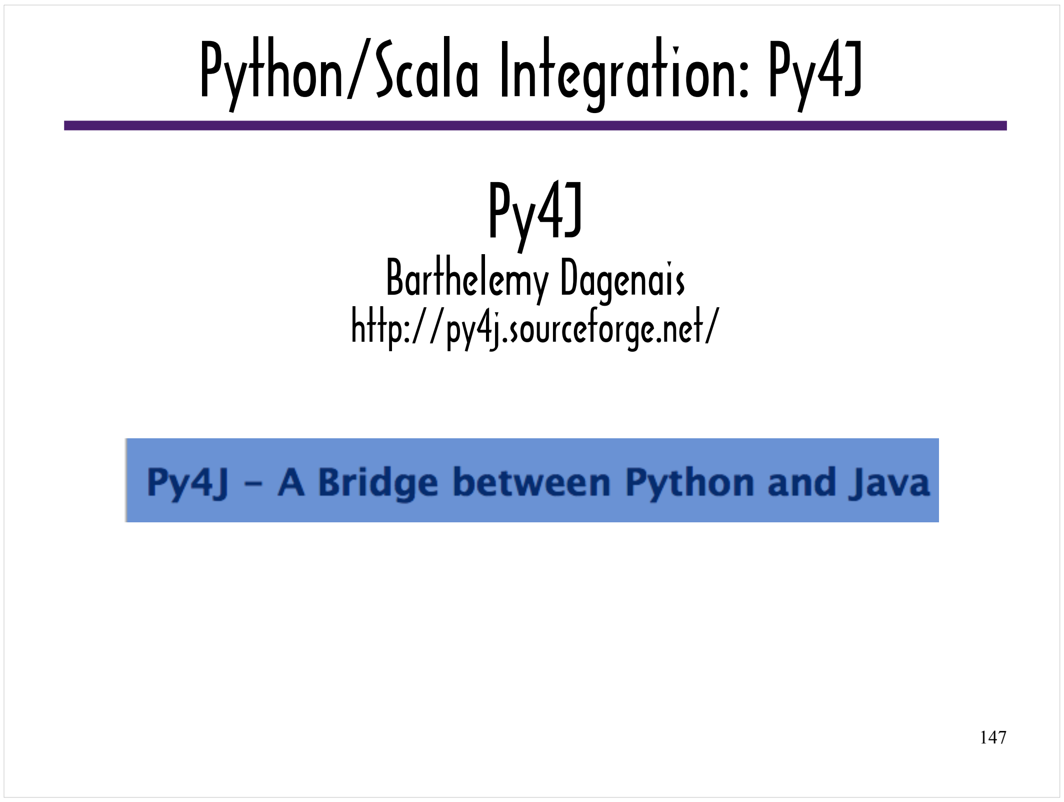
```
import staircase

staircase.initVM(staircase.CLASSPATH)

from staircase import ScalaUtils

print(ScalaUtils.ageGroupMethod(21.0) + 1)
```

145

With the few lines of initialization out of the way, you can now call use the Scala library from within Python.

## Python//Scala Interop Via Language-Neutral Protocols

This last section of this talk focuses on two projects that wrap language-neutral protocols.



## Python/Scala Integration: Py4J

Py4J

Barthelemy Dagenais

http://py4j.sourceforge.net/

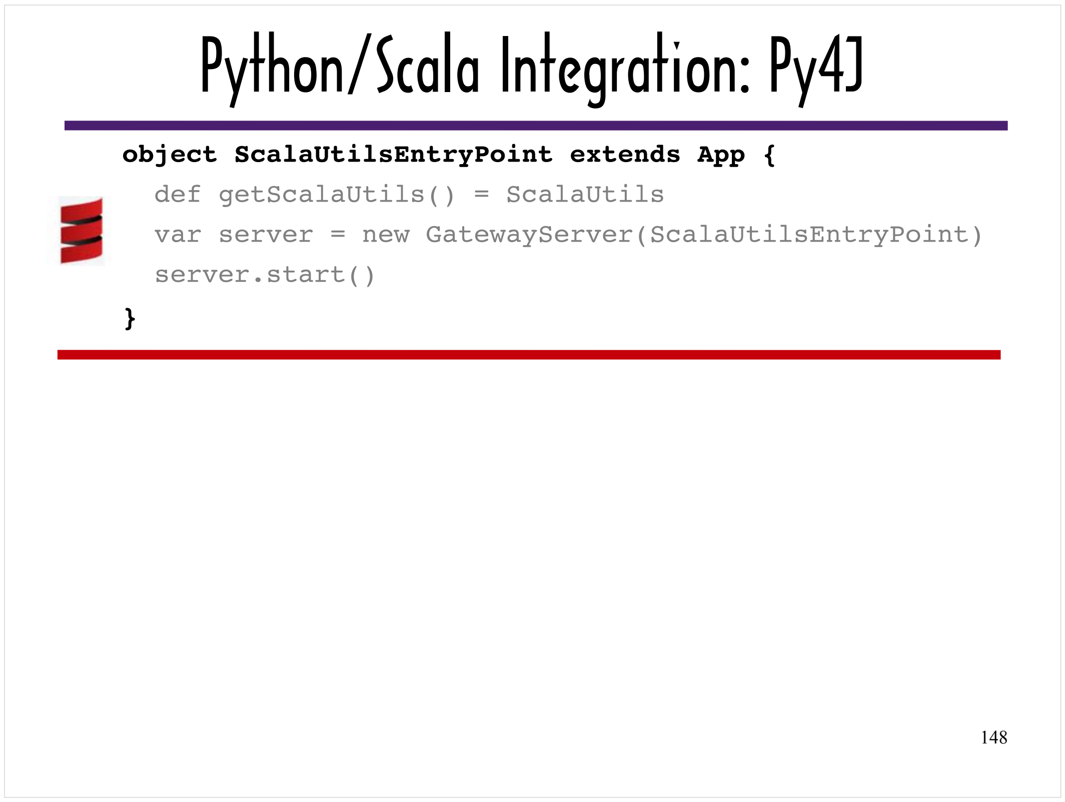
Py4J – A Bridge between Python and Java

147

Py4J, which uses Base64 over TCP for communication between a Java-based server and a Python-based client.

It was not hard to build a Scala-based server in lieu of a Java-based server using the the Py4J documentation.

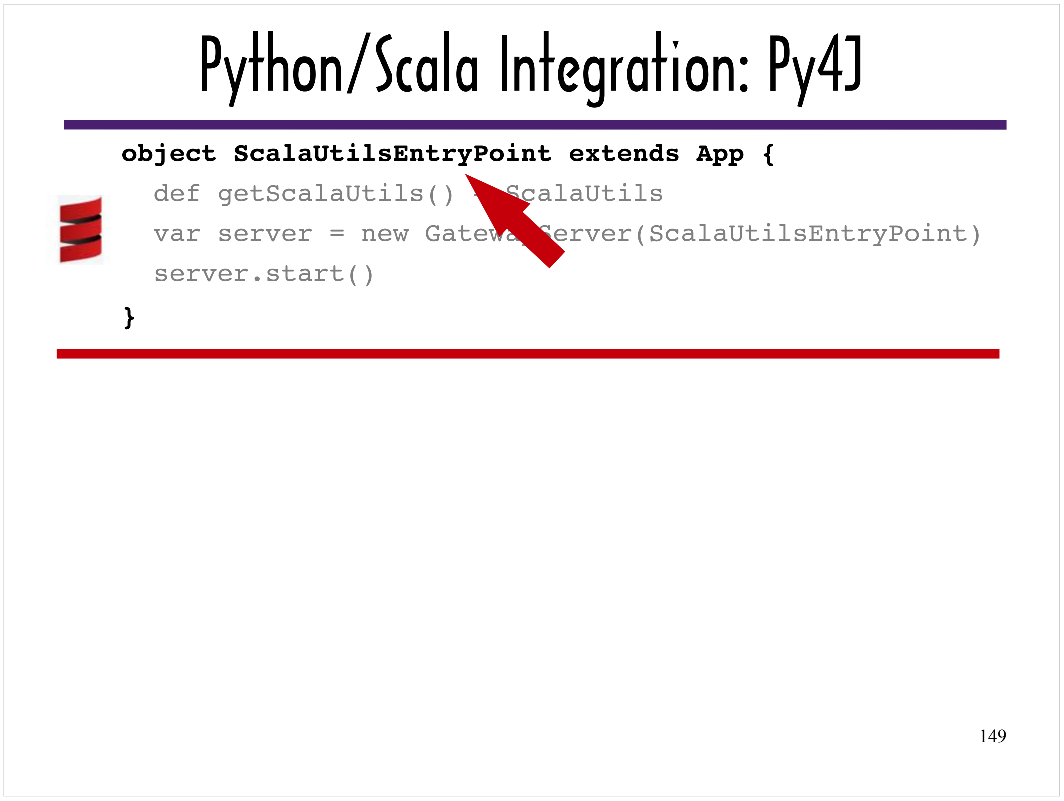
Py4J is not language-neutral in the sense that it was built specifically for Python and the JVM, but it wraps language-neutral protocols.



## Python/Scala Integration: Py4J

```
object ScalaUtilsEntryPoint extends App {
  def getScalaUtils() = ScalaUtils
  var server = new GatewayServer(ScalaUtilsEntryPoint)
  server.start()
}
```

This slide shows all of the server code, with the exception of the statement that imports the ScalaUtils micro-library, which we walked through at the start of the “Python Hosting Scala” section of this talk.

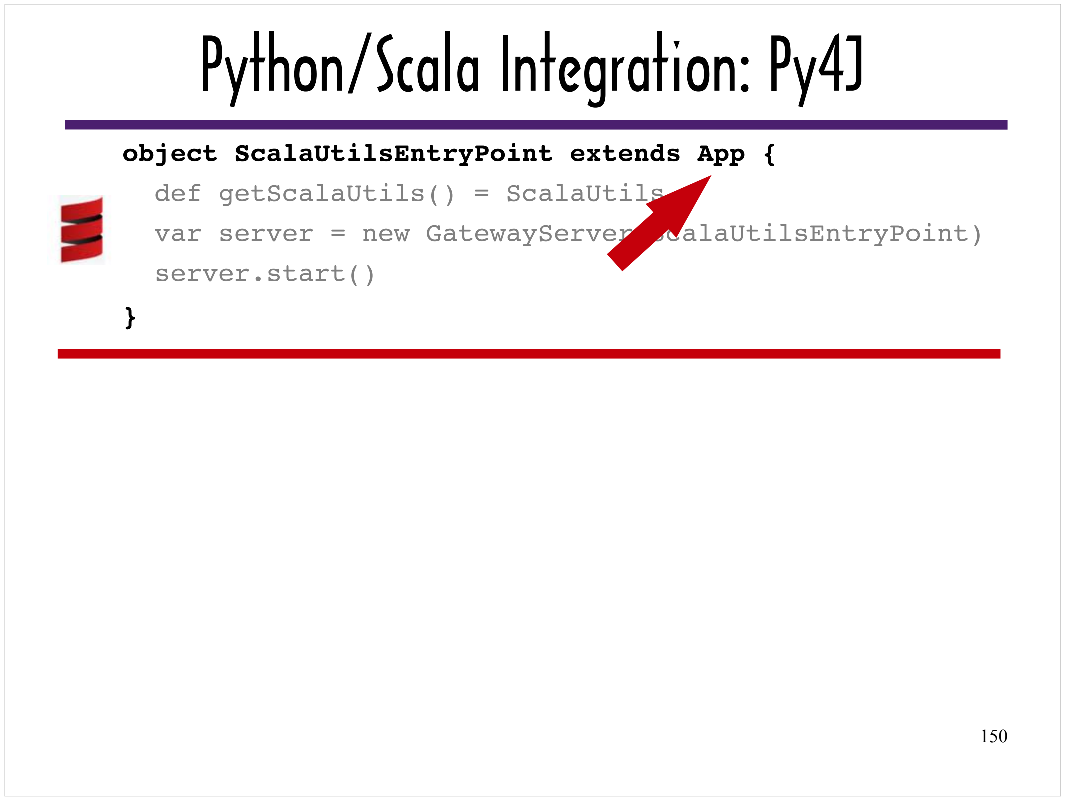


## Python/Scala Integration: Py4J

```
object ScalaUtilsEntryPoint extends App {
  def getScalaUtils() = ScalaUtils
  var server = new GatewayServer(ScalaUtilsEntryPoint)
  server.start()
}
```

A Py4J server name is typically made by appending “EntryPoint” to the name of the custom library the server provides access to (in this case, “ScalaUtils”).

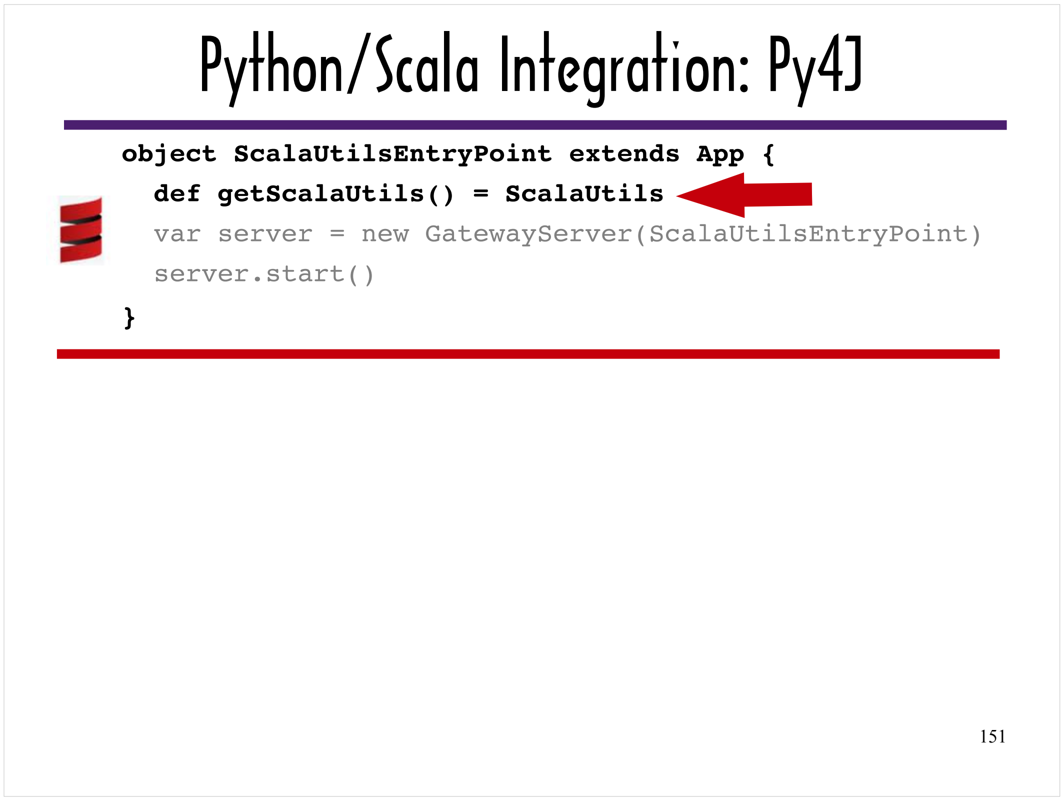
The Py4J documentation typically refers to a Py4J server as an “EntryPoint”, so I will use that convention for the remainder of the Py4J slides.



## Python/Scala Integration: Py4J

```
object ScalaUtilsEntryPoint extends App {
  def getScalaUtils() = ScalaUtils
  var server = new GatewayServer(ScalaUtilsEntryPoint)
  server.start()
}
```

Recall that the phrase **`extends App`** enables the code between the brackets to run like a script.

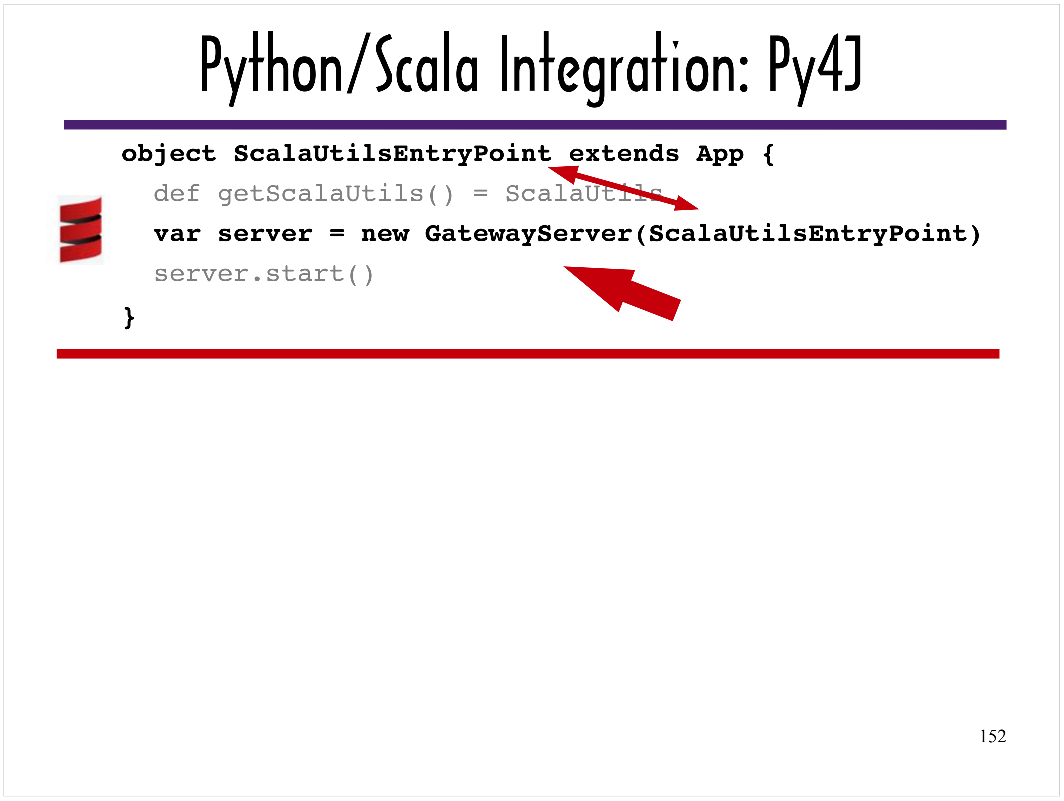


## Python/Scala Integration: Py4J

```
object ScalaUtilsEntryPoint extends App {
  def getScalaUtils() = ScalaUtils
  var server = new GatewayServer(ScalaUtilsEntryPoint)
  server.start()
}
```

151

Every Py4J EntryPoint needs a method that returns a reference to the library it provides access to, like **`getScalaUtils`**.

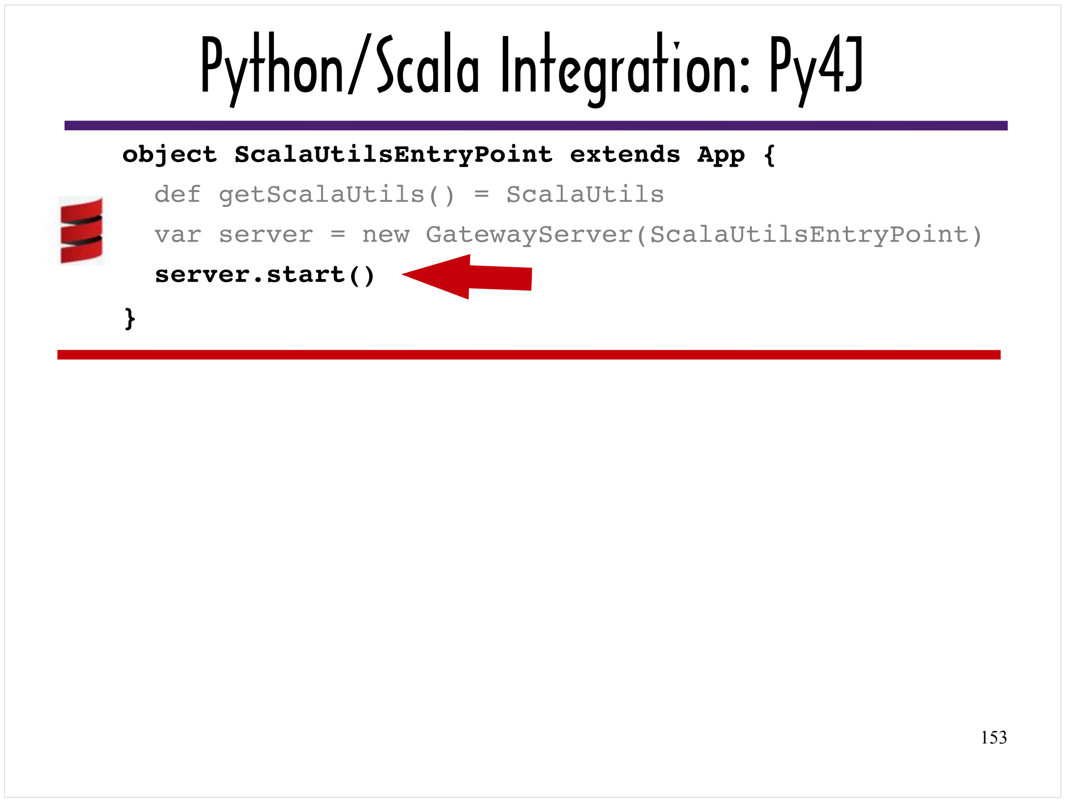


## Python/Scala Integration: Py4J

```
object ScalaUtilsEntryPoint extends App {
  def getScalaUtils() = ScalaUtils
  var server = new GatewayServer(ScalaUtilsEntryPoint)
  server.start()
}
```

A Py4J EntryPoint needs to instantiate a GatewayServer, which typically takes a reference to the EntryPoint as an argument to its constructor. Since ScalaUtilsEntryPoint is a Scala singleton object, it does not need to be instantiated.

GatewayServer is a Py4J core class that encapsulates the setting up of a TCP server, receiving Base64-encoded data over the TCP connection, and sending back a response after invoking a function in the custom library the EntryPoint provides access to (in this case ScalaUtils).



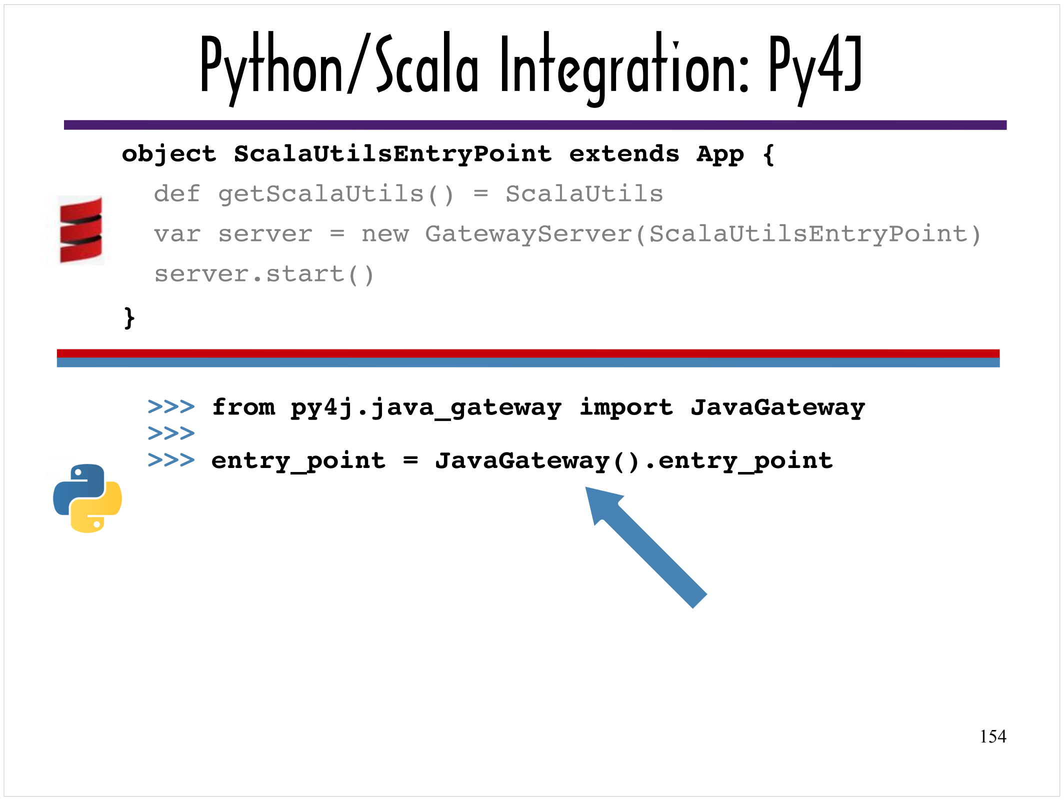
## Python/Scala Integration: Py4J

```
object ScalaUtilsEntryPoint extends App {
  def getScalaUtils() = ScalaUtils
  var server = new GatewayServer(ScalaUtilsEntryPoint)
  server.start()
}
```

153

**`GatewayServer#start`** instantiates a java.net.ServerSocket for listening for requests from a Python client.

As you will see, the Python client sends a message via TCP in the form of a method name and a collection of arguments. Py4J passes the method name and arguments to its Java Reflection engine. Using Java Reflection, Py4J can invoke a method, given the payload of the message: the name of the method and any arguments.



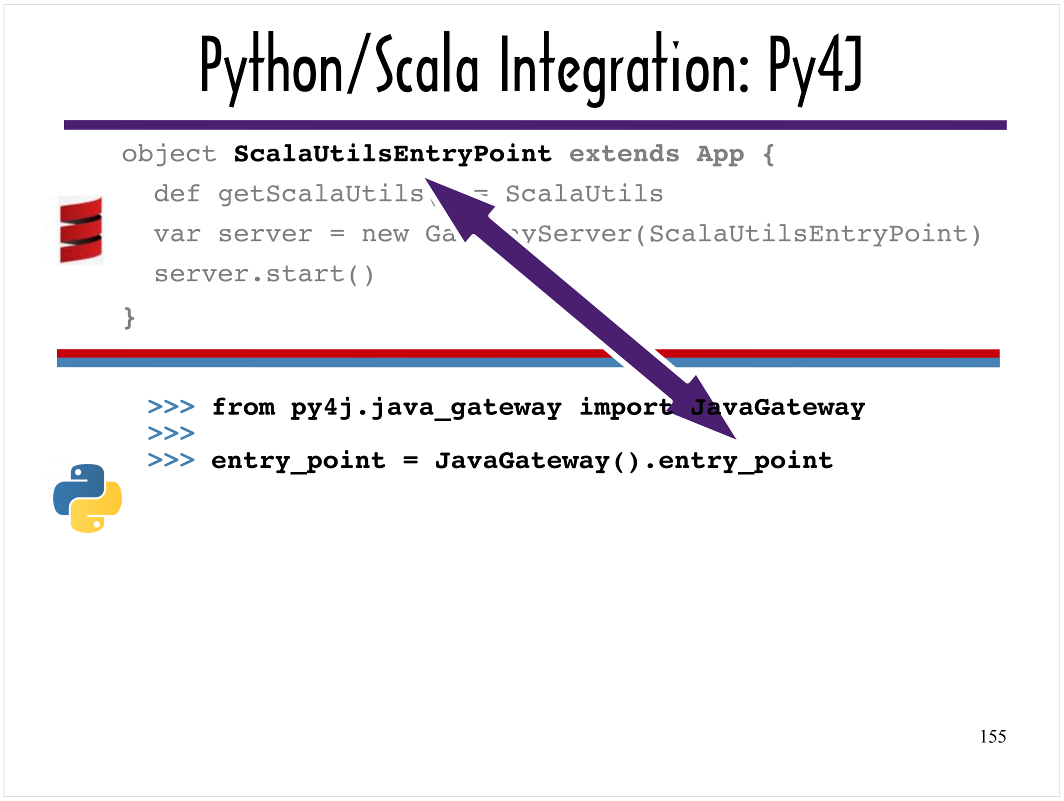
## Python/Scala Integration: Py4J

```
object ScalaUtilsEntryPoint extends App {
  def getScalaUtils() = ScalaUtils
  var server = new GatewayServer(ScalaUtilsEntryPoint)
  server.start()
}
```

```
>>> from py4j.java_gateway import JavaGateway
>>>
>>> entry_point = JavaGateway().entry_point
```

I'll use a simulated Python interactive console to show the Python client code.

The Python client needs a reference to ...



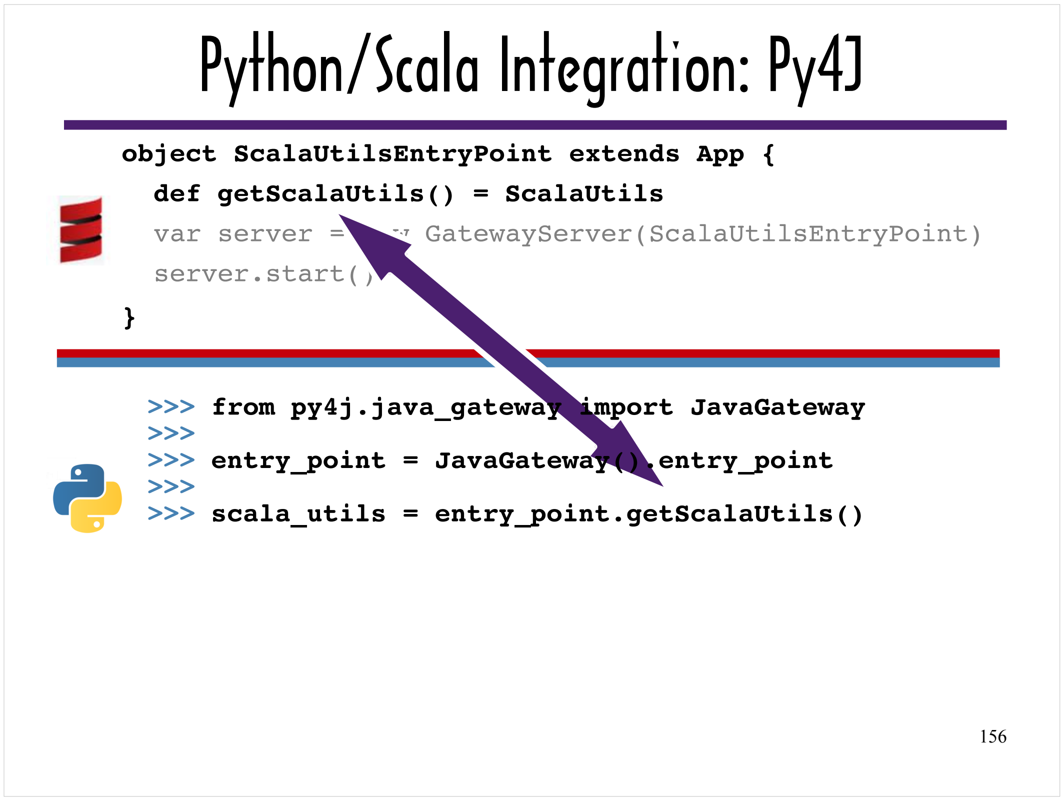
## Python/Scala Integration: Py4J

```
object ScalaUtilsEntryPoint extends App {
  def getScalaUtils() = ScalaUtils
  var server = new GatewayServer(ScalaUtilsEntryPoint)
  server.start()
}
```

```
>>> from py4j.java_gateway import JavaGateway
>>>
>>> entry_point = JavaGateway().entry_point
```

155

**...**the EntryPoint, which can be obtained by accessing the **`entry_point`** property of an instance of the Py4J core class, JavaGateway.



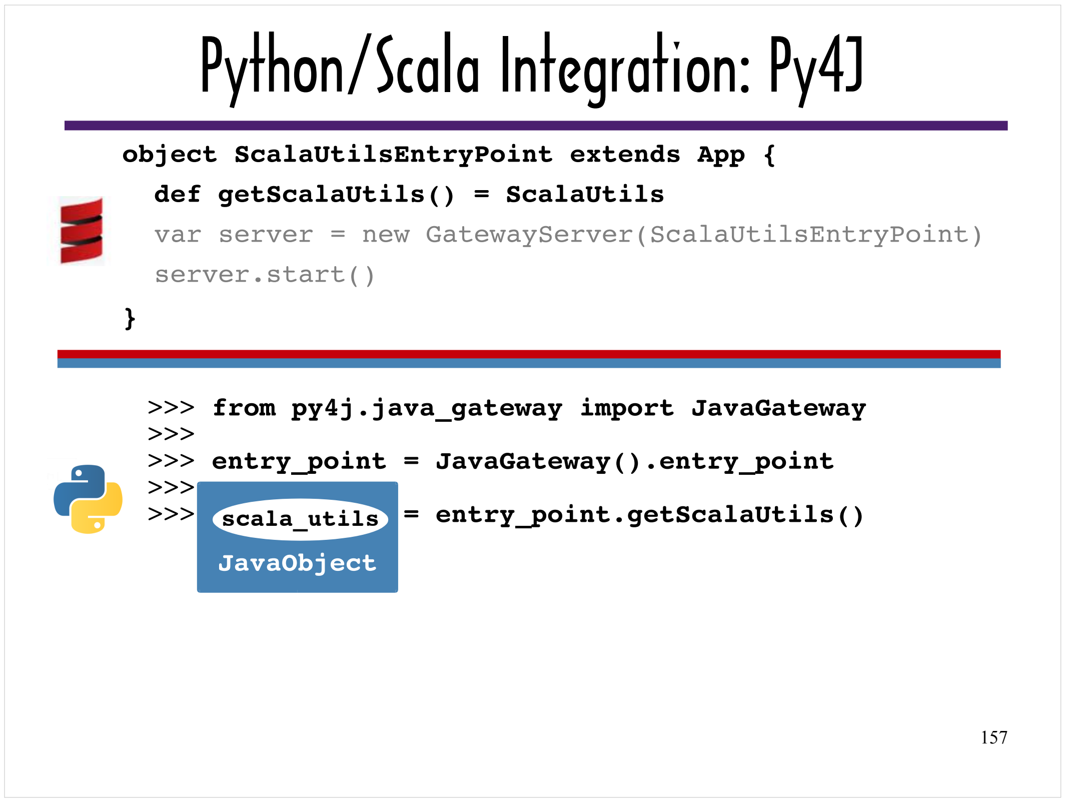
## Python/Scala Integration: Py4J

```
object ScalaUtilsEntryPoint extends App {
  def getScalaUtils() = ScalaUtils
  var server = new GatewayServer(ScalaUtilsEntryPoint)
  server.start()
}
```

```
>>> from py4j.java_gateway import JavaGateway
>>>
>>> entry_point = JavaGateway().entry_point
>>>
>>> scala_utils = entry_point.getScalaUtils()
```

The Python client can then obtain a reference to the custom Scala library that the Entry Point provides access to.

In this example, the exposed library is assigned to **`scala_utils`**.



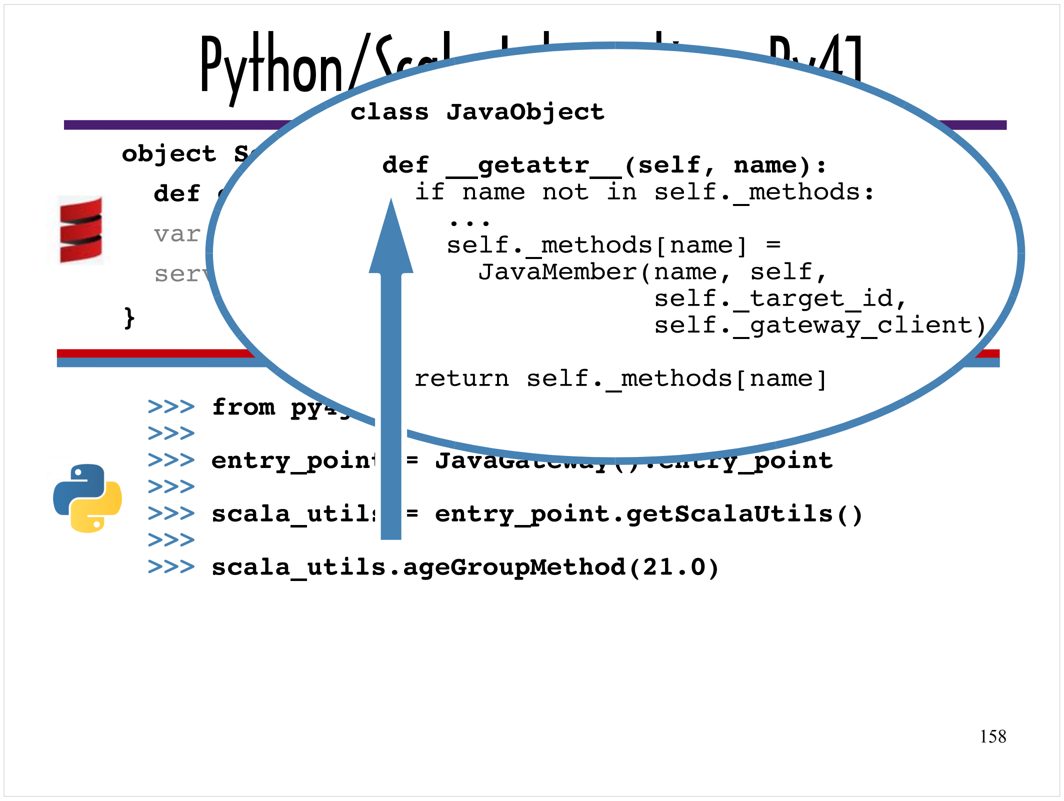
## Python/Scala Integration: Py4J

```scala
object ScalaUtilsEntryPoint extends App {
  def getScalaUtils() = ScalaUtils
  var server = new GatewayServer(ScalaUtilsEntryPoint)
  server.start()
}
```

```python
>>> from py4j.java_gateway import JavaGateway
>>>
>>> entry_point = JavaGateway().entry_point
>>>
>>> scala_utils = entry_point.getScalaUtils()
```

JavaObject

Py4J wraps the exposed library reference in a JavaObject, a Python class which ...



## Python/Scal Py4J

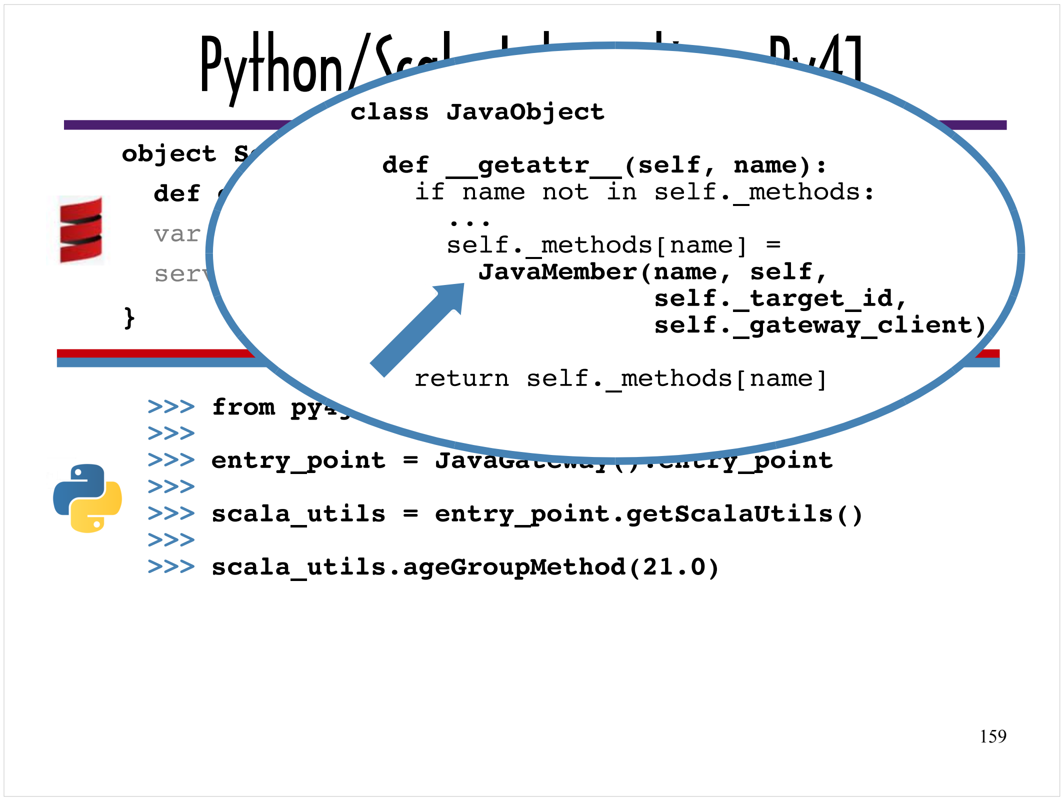
```
object S
  def
  var
  serv
}
```

```
class JavaObject

  def __getattr__(self, name):
    if name not in self._methods:
      ...
      self._methods[name] =
        JavaMember(name, self,
                   self._target_id,
                   self._gateway_client)

    return self._methods[name]
```

```
>>> from py4
>>>
>>> entry_poin  = JavaGateway().entry_point
>>>
>>> scala_util  = entry_point.getScalaUtils()
>>>
>>> scala_utils.ageGroupMethod(21.0)
```

158

… overrides **`__getattr__`** so that when the Python client tries to invoke one of the custom Scala library methods using dot notation and the Python runtime detects that the Python client is trying to invoke a method that is not defined as an attribute on the JavaObject instance ...



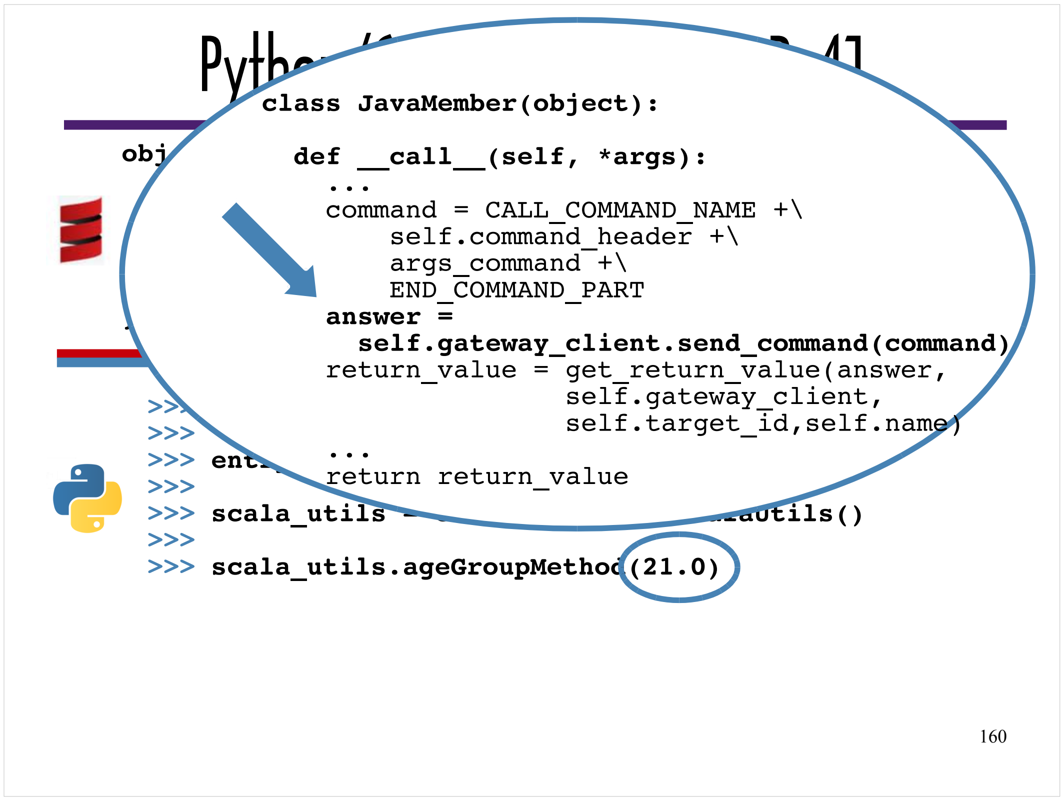
## Python/Scal Py4J

```
object S
  def
  var
  serv
}
```

```
class JavaObject

  def __getattr__(self, name):
    if name not in self._methods:
      ...
      self._methods[name] =
        JavaMember(name, self,
                   self._target_id,
                   self._gateway_client)

    return self._methods[name]
```

```
>>> from py4
>>>
>>> entry_point = JavaGateway().entry_point
>>>
>>> scala_utils = entry_point.getScalaUtils()
>>>
>>> scala_utils.ageGroupMethod(21.0)
```

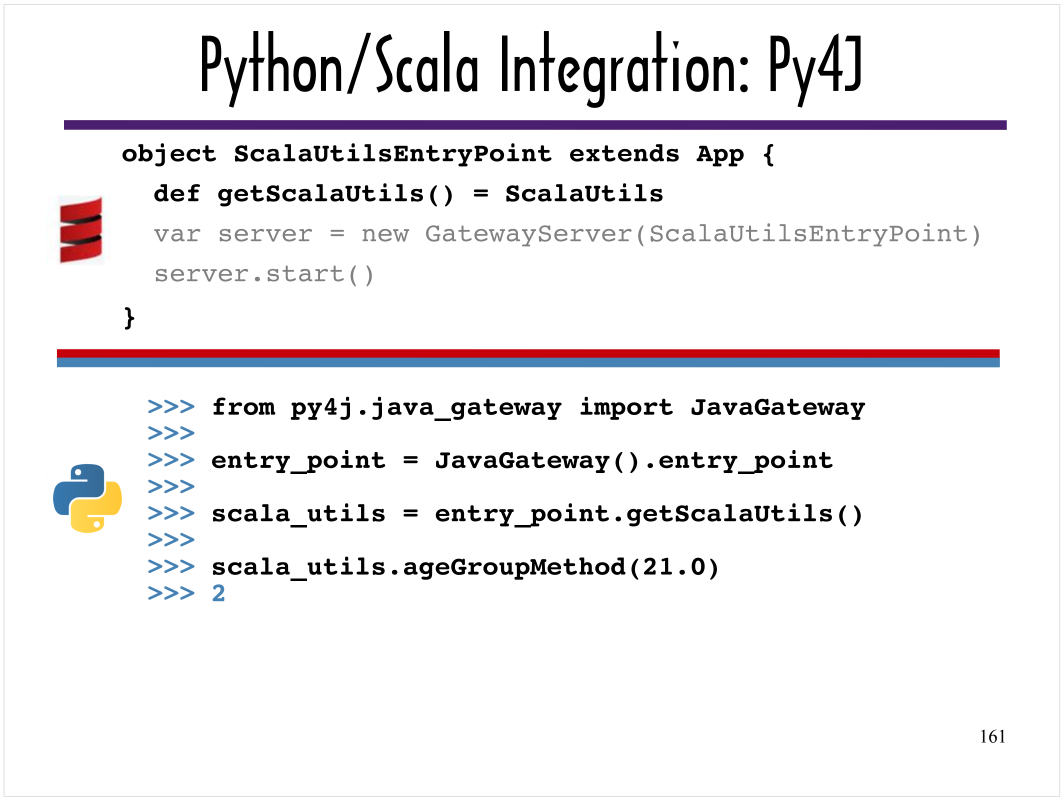
… the specified custom library method is wrapped in a JavaMember, which ...



```
class JavaMember(object):

    def __call__(self, *args):
      ...
      command = CALL_COMMAND_NAME +\
          self.command_header +\
          args_command +\
          END_COMMAND_PART
      answer =
        self.gateway_client.send_command(command)
      return_value = get_return_value(answer,
                     self.gateway_client,
                     self.target_id,self.name)
      ...
      return return_value
```

```
>>> scala_utils.ageGroupMethod(21.0)
```

… defines **`__call__`** (which is invoked when the argument list is tacked onto the **`ageGroupMethod`**) to encapsulate encoding the method identifier and arguments and sending them to the EntryPoint via TCP.

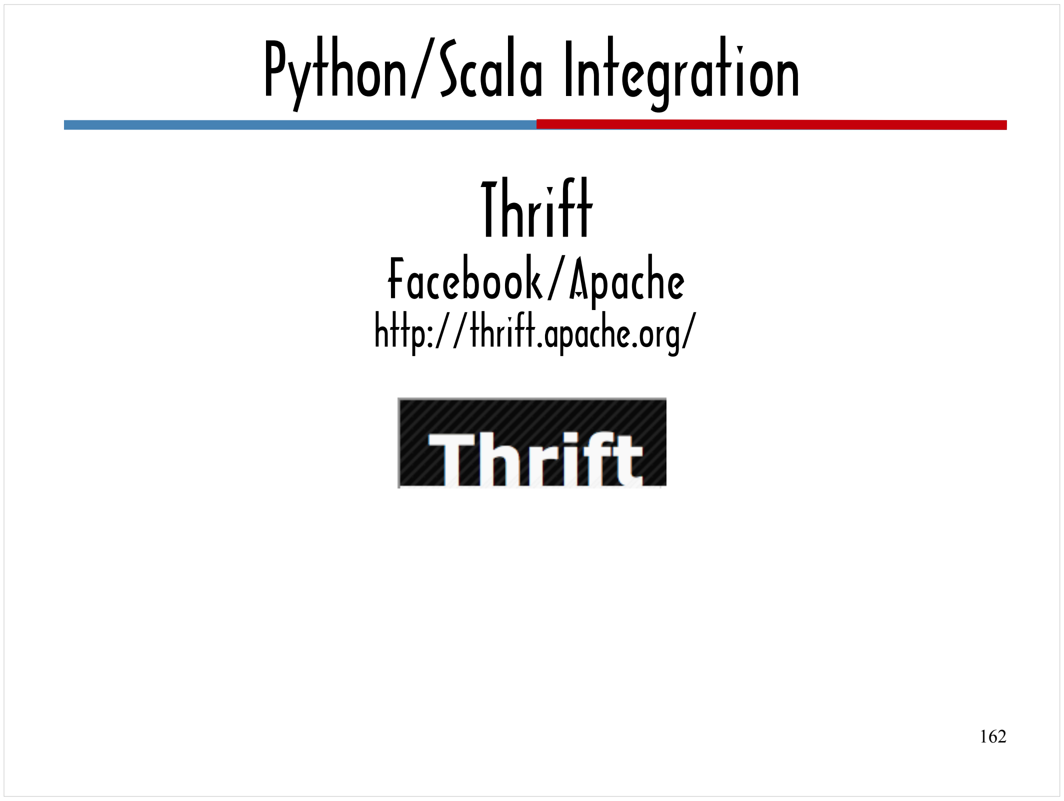


## Python/Scala Integration: Py4J

```
object ScalaUtilsEntryPoint extends App {
  def getScalaUtils() = ScalaUtils
  var server = new GatewayServer(ScalaUtilsEntryPoint)
  server.start()
}
```

```
>>> from py4j.java_gateway import JavaGateway
>>>
>>> entry_point = JavaGateway().entry_point
>>>
>>> scala_utils = entry_point.getScalaUtils()
>>>
>>> scala_utils.ageGroupMethod(21.0)
>>> 2
```

Python's **`__getattr__`** makes Py4J viable. Without it, it would not be possible to support arbitrary method calls on a Scala library. Without it, you would need to create an individual wrapper for each method, function and property in the custom Scala library.



## Python/Scala Integration

### Thrift

Facebook/Apache

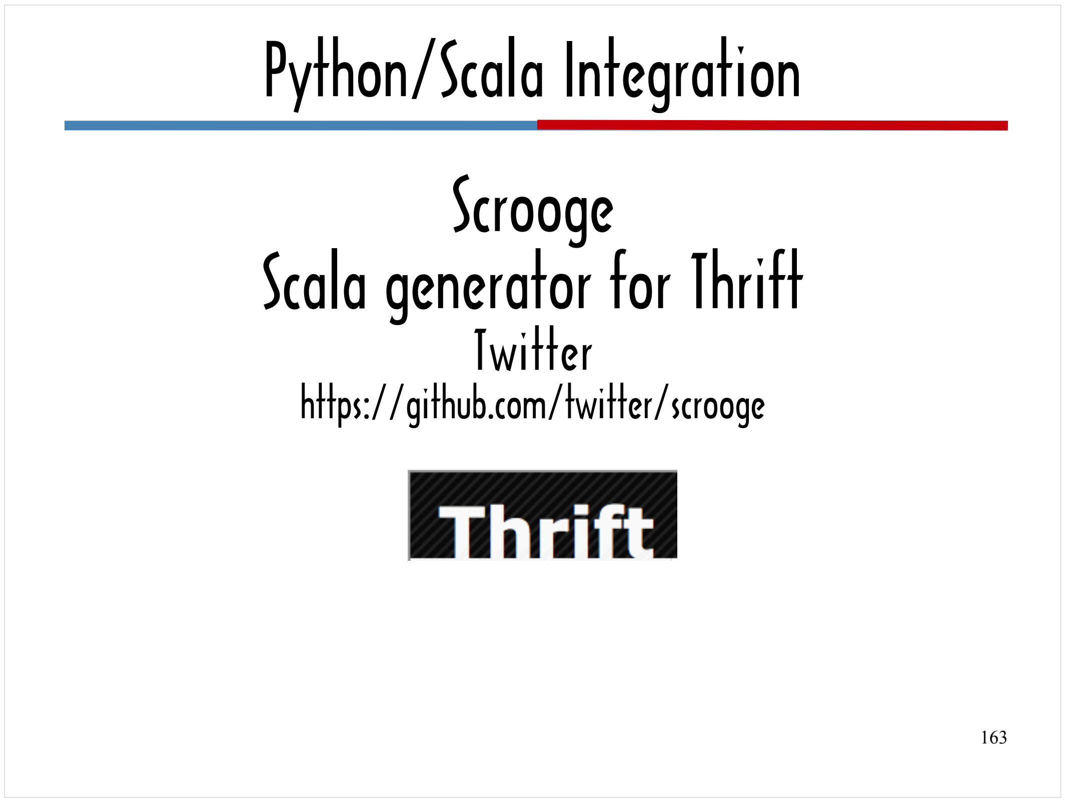
http://thrift.apache.org/

Thrift

Like Py4J, Thrift, wraps language-neutral transport and encapsulation protocols (e.g. TCP, binary and JSON), but it's also language-neutral by virtue of being based on an Interface Definition Language (IDL).

With Thrift, you use a generic IDL to specify elements of an interface, and Thrift generates client and server code in any number of different languages.

Another example of an IDL-based project is Google's Protocol Buffers (http://code.google.com/p/protobuf/)



## Python/Scala Integration

Scrooge
Scala generator for Thrift
Twitter
https://github.com/twitter/scrooge

Thrift

Thrift is packaged with generators for Python, Java, C++, Erlang, PHP, Ruby, Go, Ocaml and several other languages. It is not come with Scala generators.

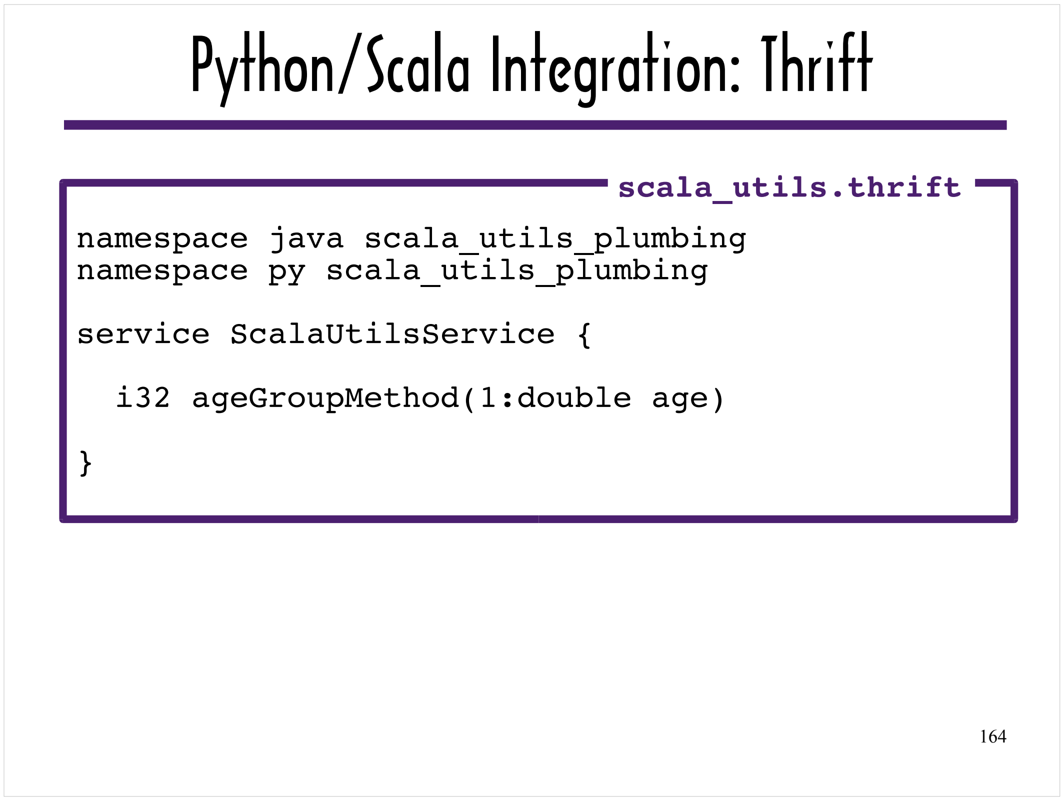
I use the Java generator in my Scala/Python interop example.

If you need to integrate Python and Scala in the real world, the Java generator may work for you, but it would be a good idea to take a look at Scrooge, a Scala generator for Thrift written by the Twitter team.

I did not use Scrooge for my example for a couple of reasons:

1. You can get a certain distance for a Scala-based project using the Thrift Java generators. Twitter used Thrift's Java generators in conjunction with Scala in production prior to completing their Thrift Scala generator.

2. it is dependent on some external Twitter libraries and since I want to focus on Thrift as an IDL-based approach to polyglot systems and not the relative merits of the specific protocol and transport layer options packaged with Thrift, I would rather show what Thrift does by default than take a detour to cover Twitter's enhancements.



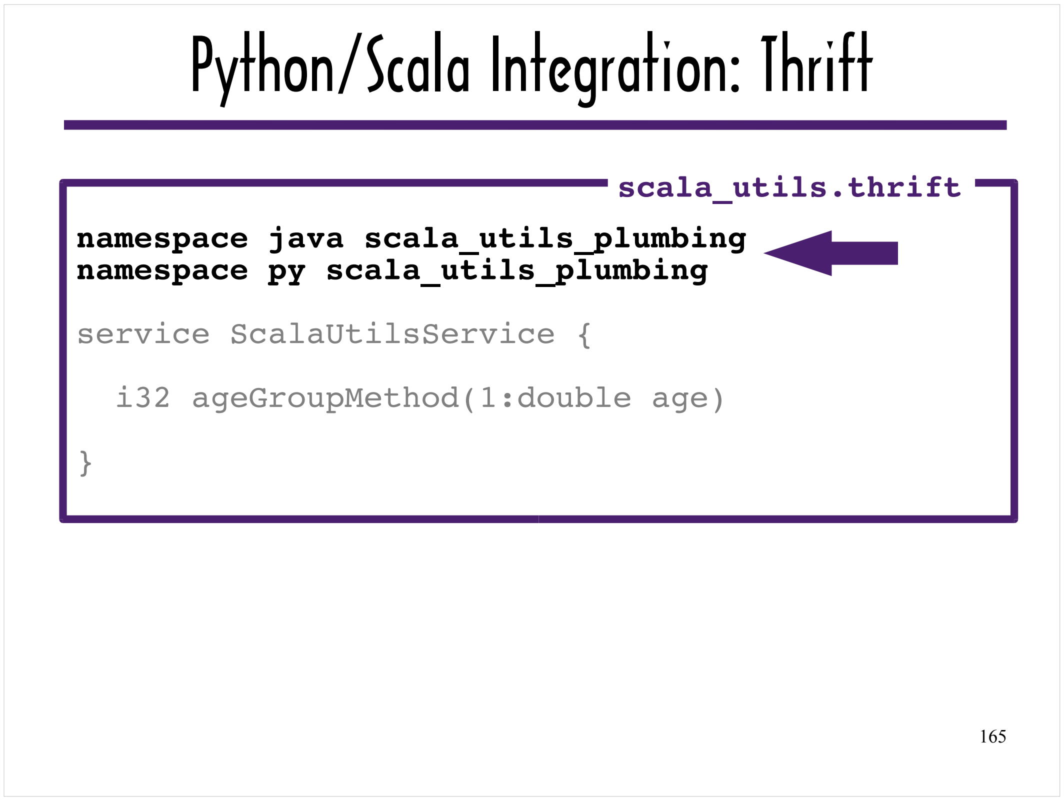
## Python/Scala Integration: Thrift

**scala_utils.thrift**

```
namespace java scala_utils_plumbing
namespace py scala_utils_plumbing

service ScalaUtilsService {

  i32 ageGroupMethod(1:double age)

}
```

This slide represents **`scala_utils.thrift`**, a Thrift IDL file  that specifies the interface for **`ScalaUtilsServer#ageGroupMethod`**, which I'm going to back with code from the Scala micro-library I have used for many other examples.

Based on this IDL file. Thrift will generate both server and client code for Python as well as Java, but I'm only going to walk through the code for a system that is comprised of a Scala-based server and a Python-based client.



## Python/Scala Integration: Thrift

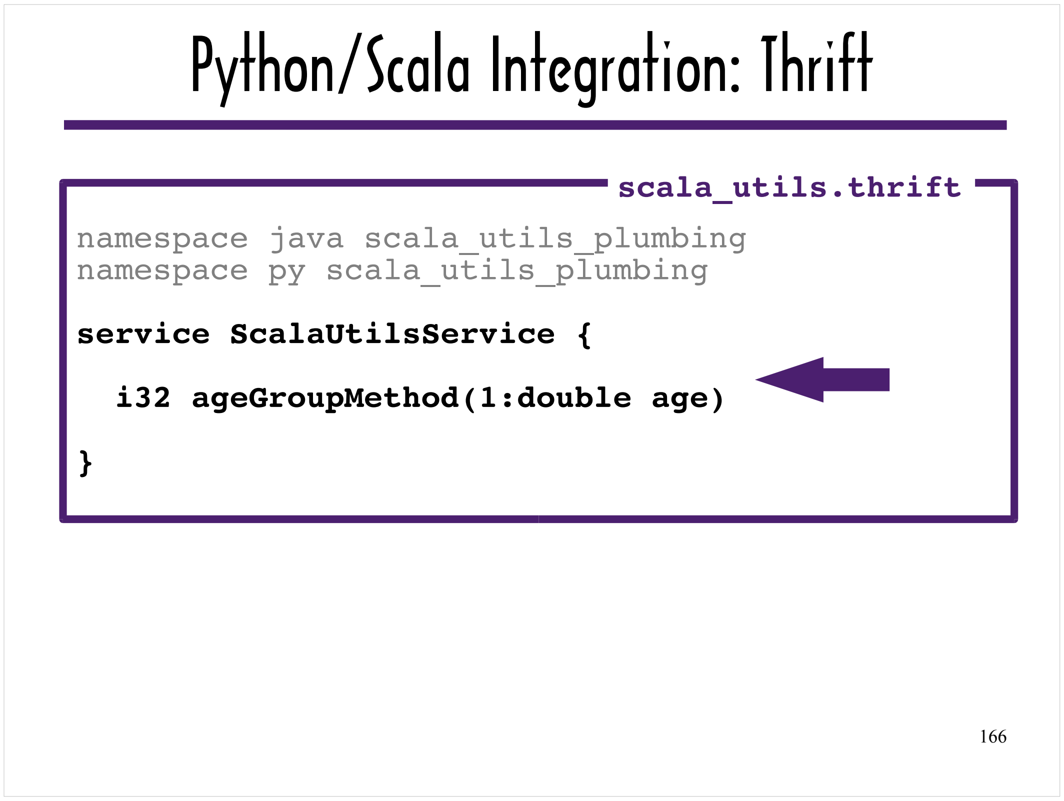
scala_utils.thrift

```
namespace java scala_utils_plumbing
namespace py scala_utils_plumbing

service ScalaUtilsService {

  i32 ageGroupMethod(1:double age)

}
```

Thrift uses the namespace keyword differently for different languages.

Based on the lines highlighted, Thrift will use **`scala_utils_plumbing`** as the package name for the Java classes and the directory name for the generated Python code under **`gen-py`**.

I chose to use **`scala_utils_plumbing`** for both Python and the Scala because the service my sample system is built on top of is written in Scala and I think the generated Scala server code and the generated Python client code serve a plumbing for the Scala service.



## Python/Scala Integration: Thrift

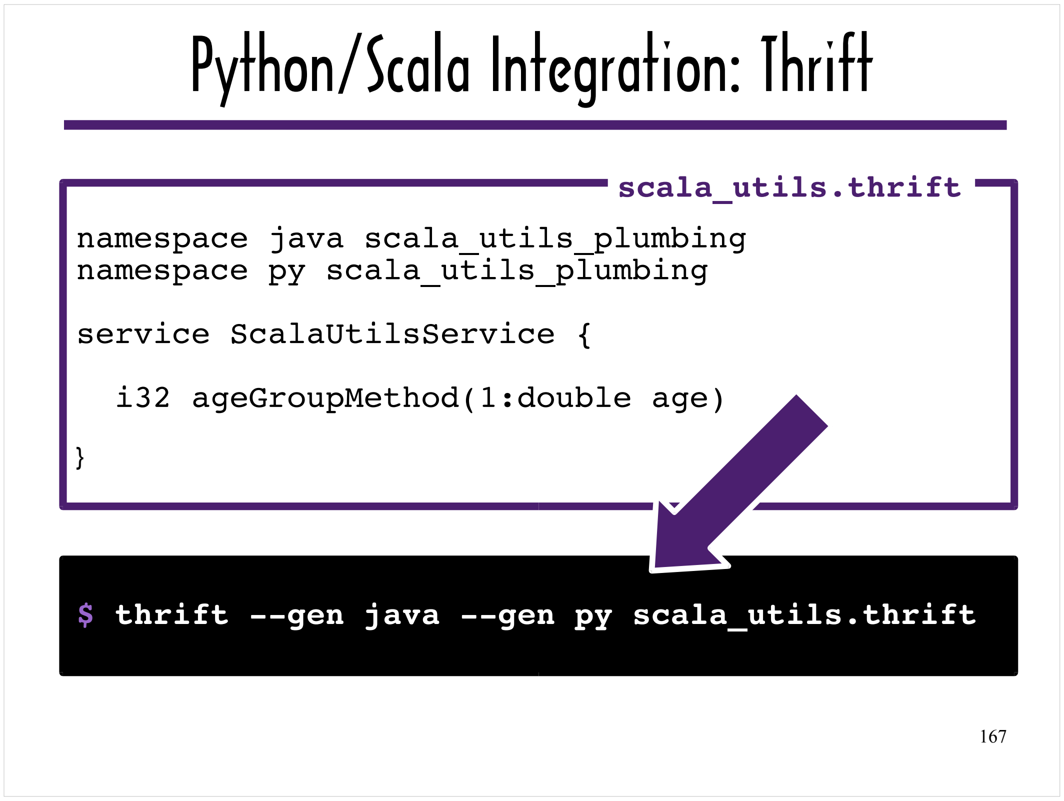
scala_utils.thrift

```
namespace java scala_utils_plumbing
namespace py scala_utils_plumbing

service ScalaUtilsService {

  i32 ageGroupMethod(1:double age)

}
```

Thrift will use the service name (in this case, **`ScalaUtilsService`**) as the top-level Java class name for the generated outer class that will contain sub-classes that encapsulate protocol and transport layer details. The same name will be used as the module name for the corresponding Python module.

In this Thrift IDL file, **`i32`** indicates that the **`ageGroupMethod`** will return a 32-Bit integer.

In Thrift IDL, the argument list for a service must include the number and type of argument(s). In this case **`1:double`** is specified.



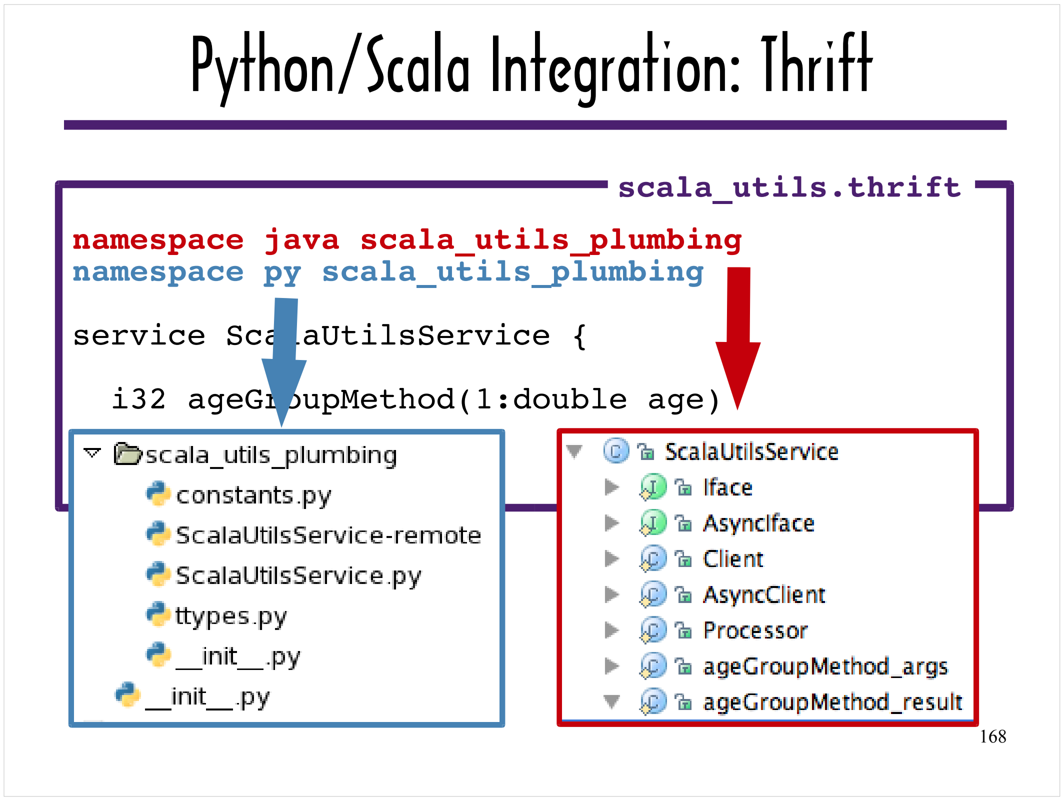
## Python/Scala Integration: Thrift

scala_utils.thrift

```
namespace java scala_utils_plumbing
namespace py scala_utils_plumbing

service ScalaUtilsService {

  i32 ageGroupMethod(1:double age)

}
```

```
$ thrift --gen java --gen py scala_utils.thrift
```

The arrow is pointing to options (**`--gen`** followed by languages specifiers) and arguments (the name of the **`.thrift`** IDL file) passed to the **`thrift`** command to trigger the Thrift generators for Java and Python.



## Python/Scala Integration: Thrift

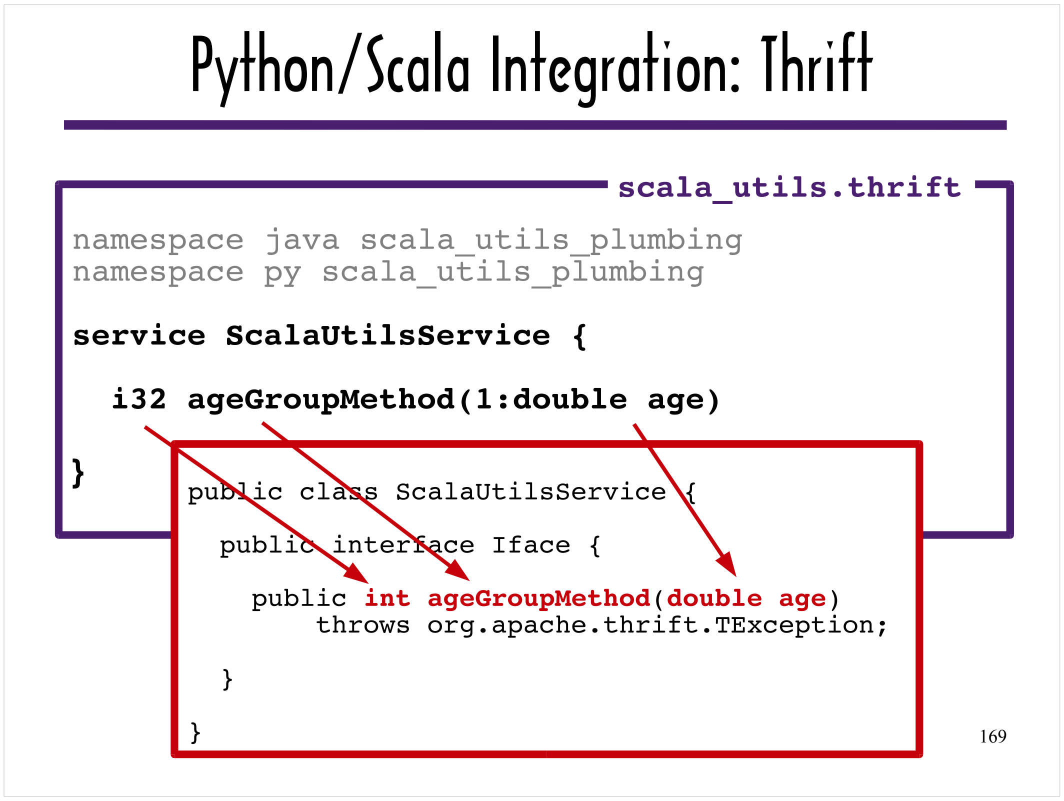
scala_utils.thrift

```
namespace java scala_utils_plumbing
namespace py scala_utils_plumbing

service ScalaUtilsService {

  i32 ageGroupMethod(1:double age)
```

```
▽ scala_utils_plumbing
    constants.py
    ScalaUtilsService-remote
    ScalaUtilsService.py
    ttypes.py
    __init__.py
  __init__.py
```

```
▼ ScalaUtilsService
  ▶ Iface
  ▶ AsyncIface
  ▶ Client
  ▶ AsyncClient
  ▶ Processor
  ▶ ageGroupMethod_args
  ▼ ageGroupMethod_result
```

Here's one view of the files Thrift generates. The Python files are shown in a Wingware IDE project in the blue-bordered box. The Java sub-classes and Interfaces are show in the JD=GUI Java class viewer in the red-bordered box.



## Python/Scala Integration: Thrift

scala_utils.thrift

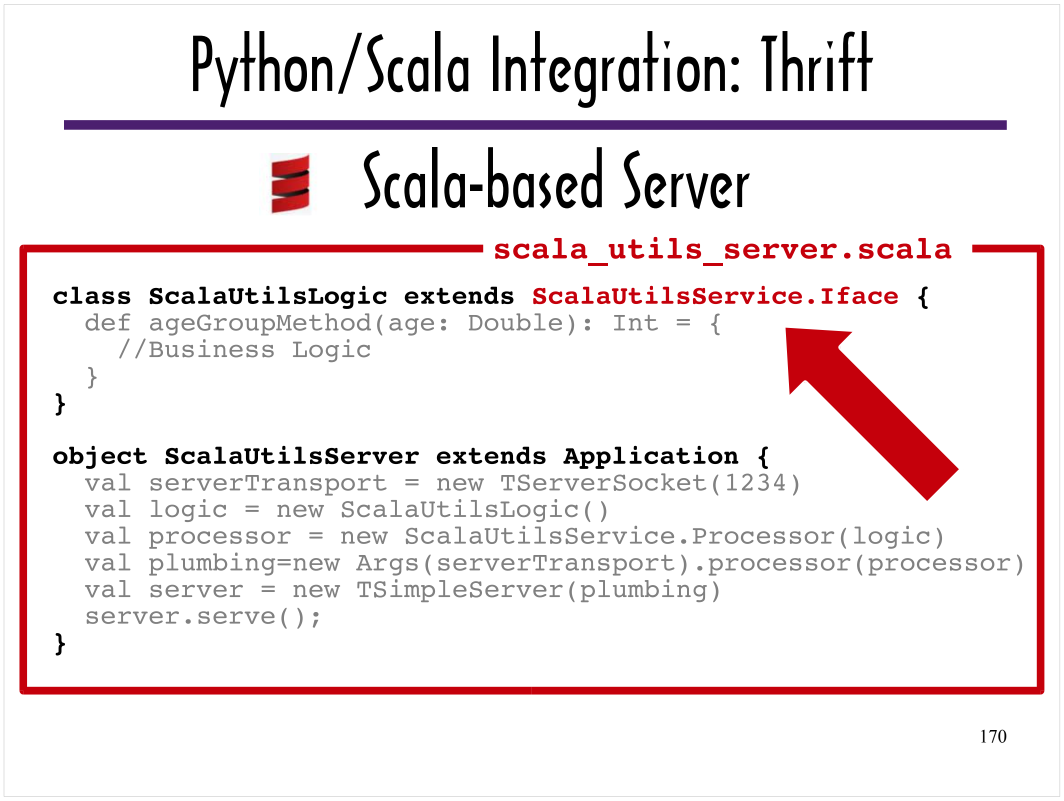
```
namespace java scala_utils_plumbing
namespace py scala_utils_plumbing

service ScalaUtilsService {

  i32 ageGroupMethod(1:double age)

}
```

```
public class ScalaUtilsService {

  public interface Iface {

    public int ageGroupMethod(double age)
        throws org.apache.thrift.TException;

  }

}
```

The red box shows the source code for the Java interface Thrift generates for the **ageGroupMethod** service, ScalaUtilsService.Iface.

As I explained when I talked about scalafying a Python function, a Java interface is similar to a Scala trait (or a Python abstract base class), but it can only include abstract method declarations. In the next slide I will show how the Scala-based server I wrote uses the Iface Java Interface.

I left the IDL source in the background and used arrows to show how the generated code reflects the IDL source.



# Python/Scala Integration: Thrift

## Scala-based Server

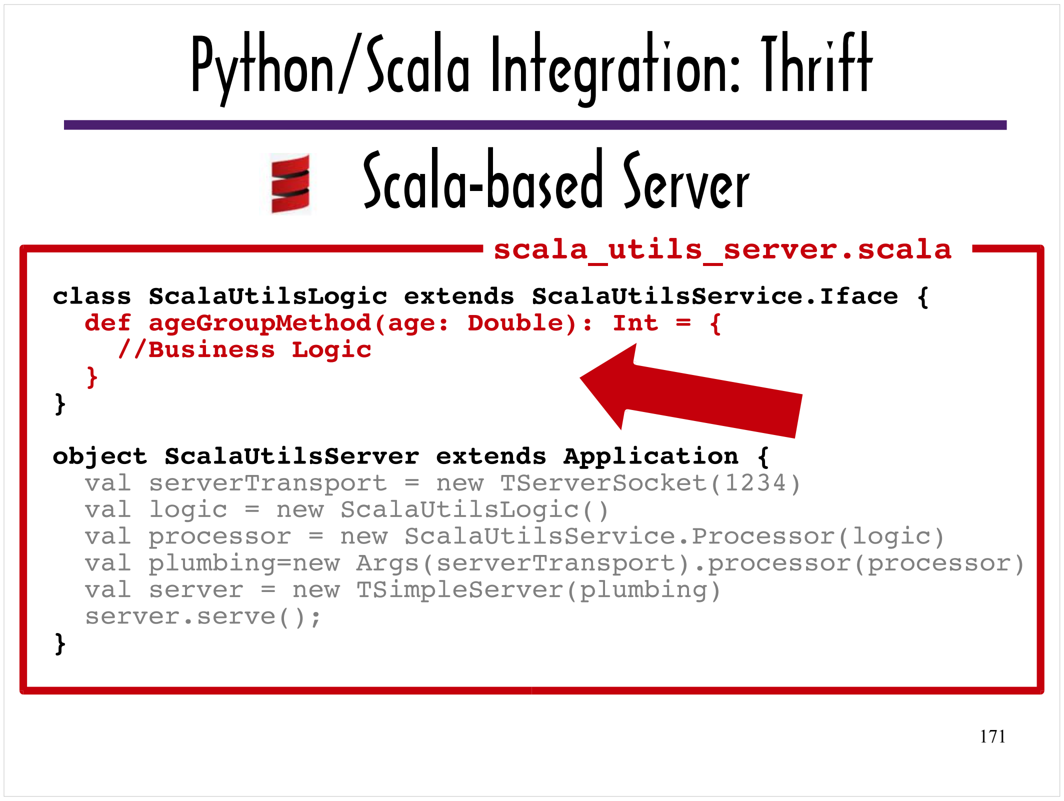
scala_utils_server.scala

```
class ScalaUtilsLogic extends ScalaUtilsService.Iface {
  def ageGroupMethod(age: Double): Int = {
    //Business Logic
  }
}

object ScalaUtilsServer extends Application {
  val serverTransport = new TServerSocket(1234)
  val logic = new ScalaUtilsLogic()
  val processor = new ScalaUtilsService.Processor(logic)
  val plumbing=new Args(serverTransport).processor(processor)
  val server = new TSimpleServer(plumbing)
  server.serve();
}
```

170

Here's a first look at the server I wrote in Scala. It uses a combination of Thrift core classes and custom classes generated by Thrift in accordance with the IDL file I created.

The runnable server, ScalaUtilsServer, references ScalaUtilsLogic, a Scala class that provides an implementation for the interface I covered in the previous slide, ScalaUtilsService.Iface.



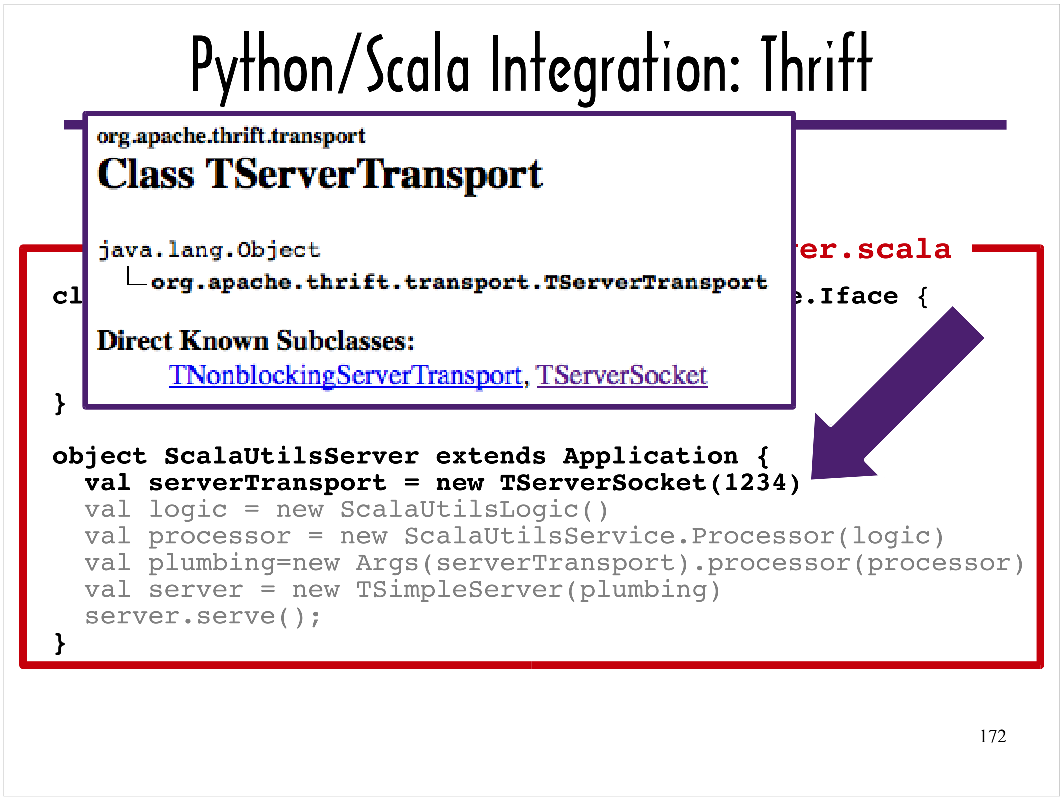
## Python/Scala Integration: Thrift

### Scala-based Server

scala_utils_server.scala

```
class ScalaUtilsLogic extends ScalaUtilsService.Iface {
  def ageGroupMethod(age: Double): Int = {
    //Business Logic
  }
}

object ScalaUtilsServer extends Application {
  val serverTransport = new TServerSocket(1234)
  val logic = new ScalaUtilsLogic()
  val processor = new ScalaUtilsService.Processor(logic)
  val plumbing=new Args(serverTransport).processor(processor)
  val server = new TSimpleServer(plumbing)
  server.serve();
}
```

I represented the Scala source code for **`ageGroupMethod`** with a comment since I went over it in detail at the beginning of the “Python Hosting Scala” section of this slide deck, and I wanted to keep this slide uncluttered.



## Python/Scala Integration: Thrift

org.apache.thrift.transport

**Class TServerTransport**

```
java.lang.Object
  └org.apache.thrift.transport.TServerTransport
```

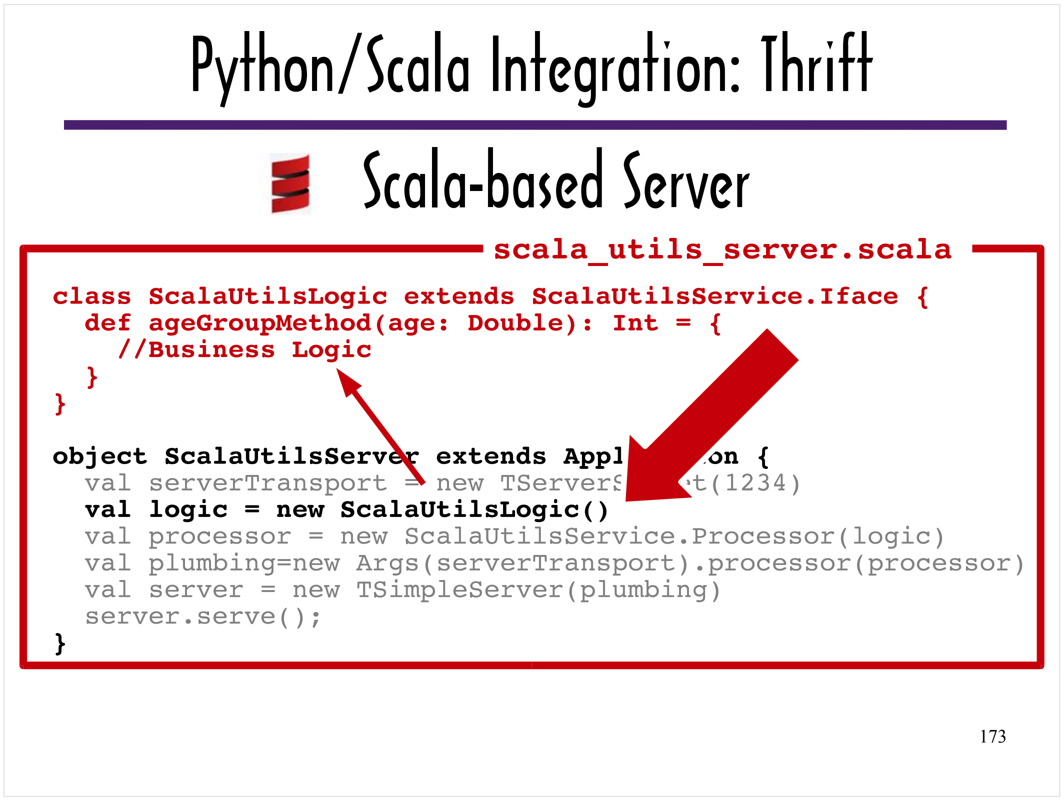
**Direct Known Subclasses:**
TNonblockingServerTransport, TServerSocket

er.scala

```
cl                                        e.Iface {

}

object ScalaUtilsServer extends Application {
  val serverTransport = new TServerSocket(1234)
  val logic = new ScalaUtilsLogic()
  val processor = new ScalaUtilsService.Processor(logic)
  val plumbing=new Args(serverTransport).processor(processor)
  val server = new TSimpleServer(plumbing)
  server.serve();
}
```

This JavaDoc shows that I can use one of two Thrift core classes, TNonblockingServerTransport or TServerSocket for client/server communication.

I chose TServerSocket, and I am passing it the port number 1234.



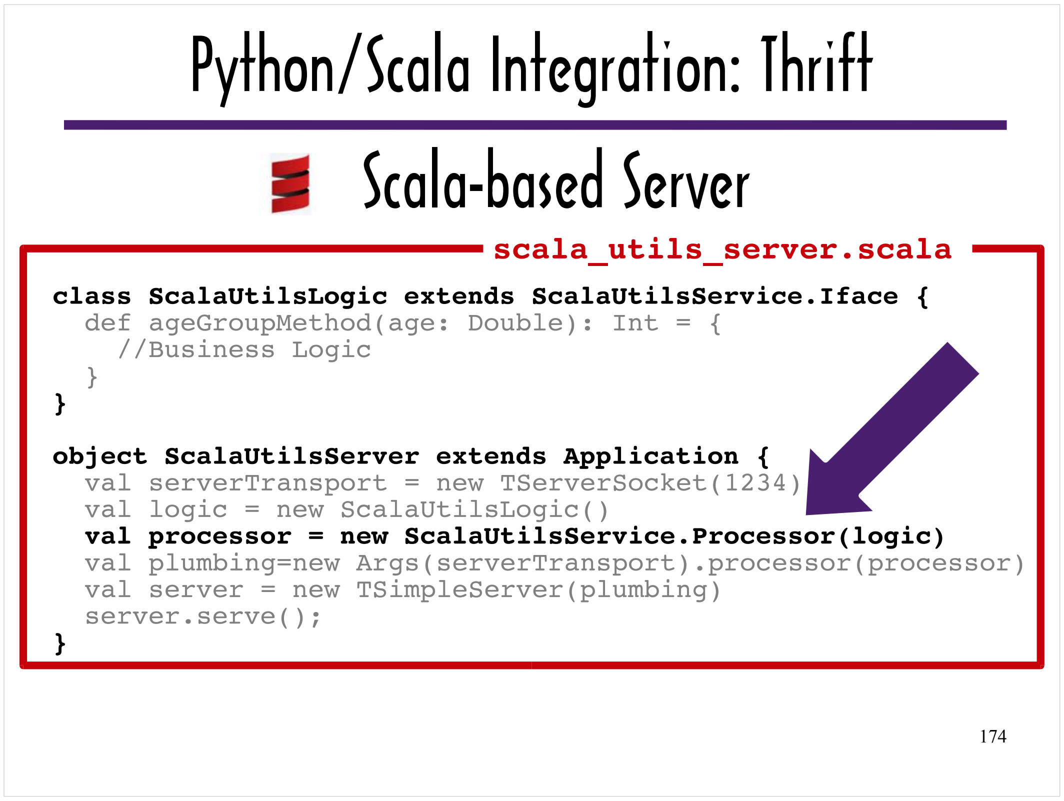
## Python/Scala Integration: Thrift

### Scala-based Server

scala_utils_server.scala

```
class ScalaUtilsLogic extends ScalaUtilsService.Iface {
  def ageGroupMethod(age: Double): Int = {
    //Business Logic
  }
}

object ScalaUtilsServer extends Appl      on {
  val serverTransport = new TServerS    t(1234)
  val logic = new ScalaUtilsLogic()
  val processor = new ScalaUtilsService.Processor(logic)
  val plumbing=new Args(serverTransport).processor(processor)
  val server = new TSimpleServer(plumbing)
  server.serve();
}
```

The “business logic” is encapsulated in the instance of ScalaUtilsLogic assigned to the variable, **`logic`**.



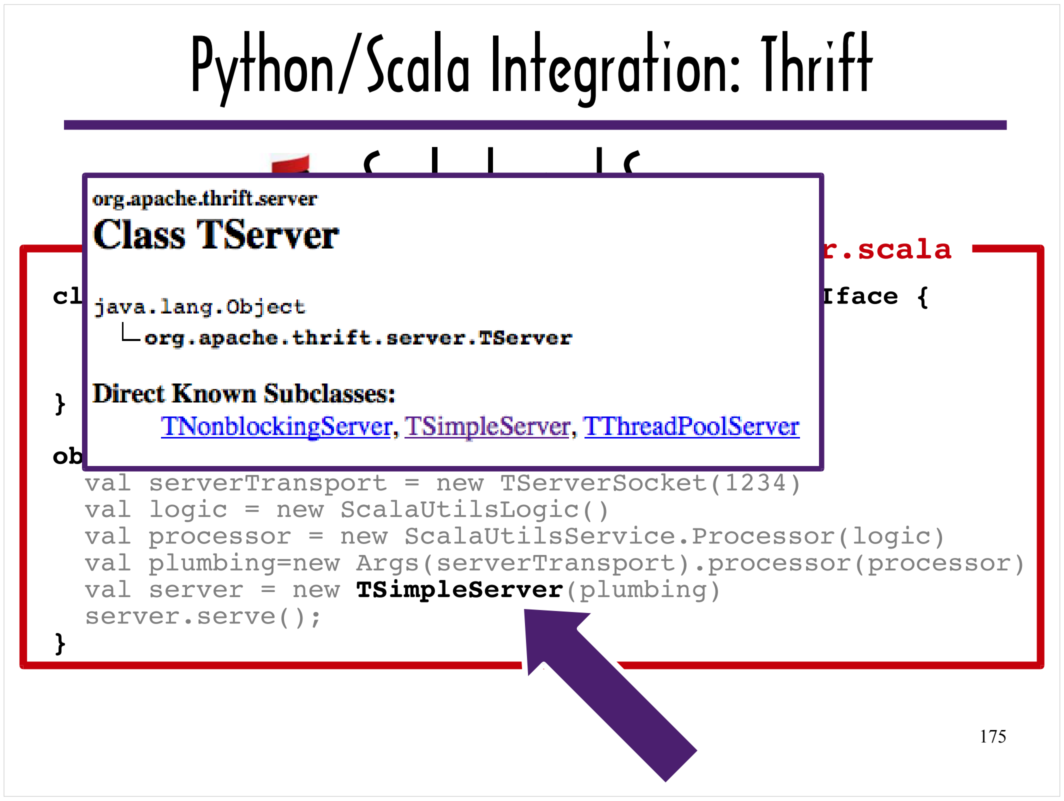
## Python/Scala Integration: Thrift

### Scala-based Server

**scala_utils_server.scala**

```
class ScalaUtilsLogic extends ScalaUtilsService.Iface {
  def ageGroupMethod(age: Double): Int = {
    //Business Logic
  }
}

object ScalaUtilsServer extends Application {
  val serverTransport = new TServerSocket(1234)
  val logic = new ScalaUtilsLogic()
  val processor = new ScalaUtilsService.Processor(logic)
  val plumbing=new Args(serverTransport).processor(processor)
  val server = new TSimpleServer(plumbing)
  server.serve();
}
```

ScalaUtilsService.Processor is a Thrift-generated class.



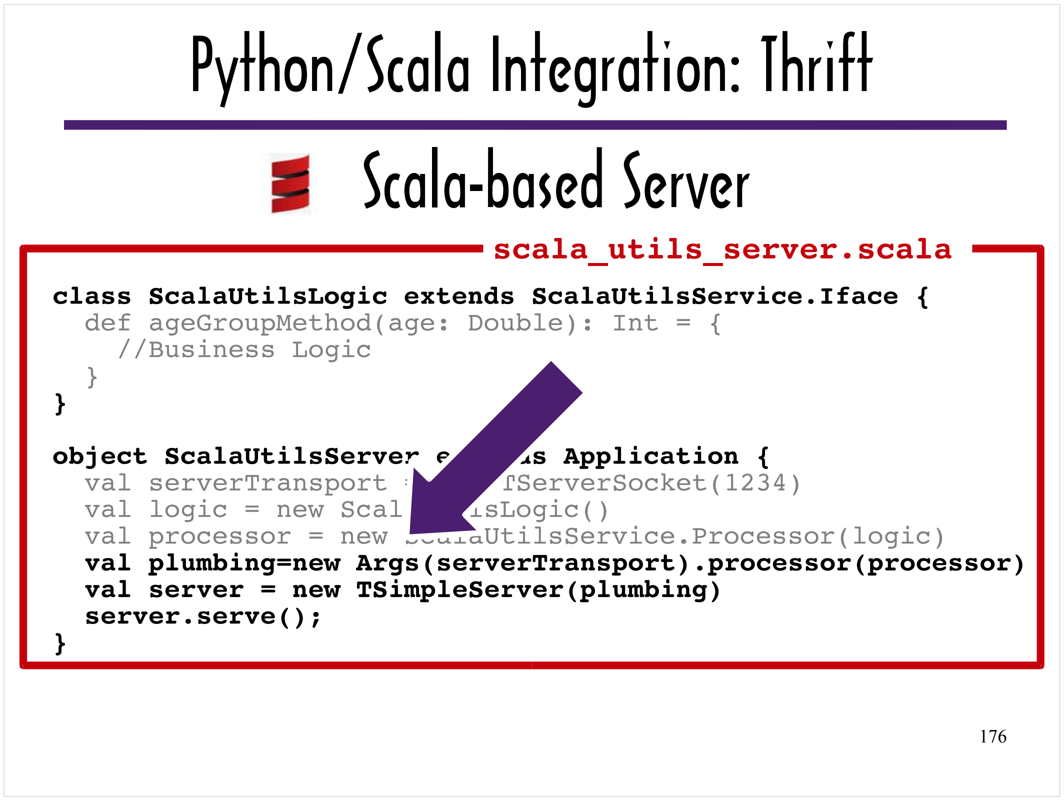
## Python/Scala Integration: Thrift

```
org.apache.thrift.server
Class TServer

java.lang.Object
  └org.apache.thrift.server.TServer

Direct Known Subclasses:
    TNonblockingServer, TSimpleServer, TThreadPoolServer
```

```
r.scala
cl                                                    Iface {

}
ob
  val serverTransport = new TServerSocket(1234)
  val logic = new ScalaUtilsLogic()
  val processor = new ScalaUtilsService.Processor(logic)
  val plumbing=new Args(serverTransport).processor(processor)
  val server = new TSimpleServer(plumbing)
  server.serve();
}
```

175

As you can see from this JavaDoc, you can choose to use a non-blocking server, thread pool server (which leverages Java's concurrency management features) or a basic server.



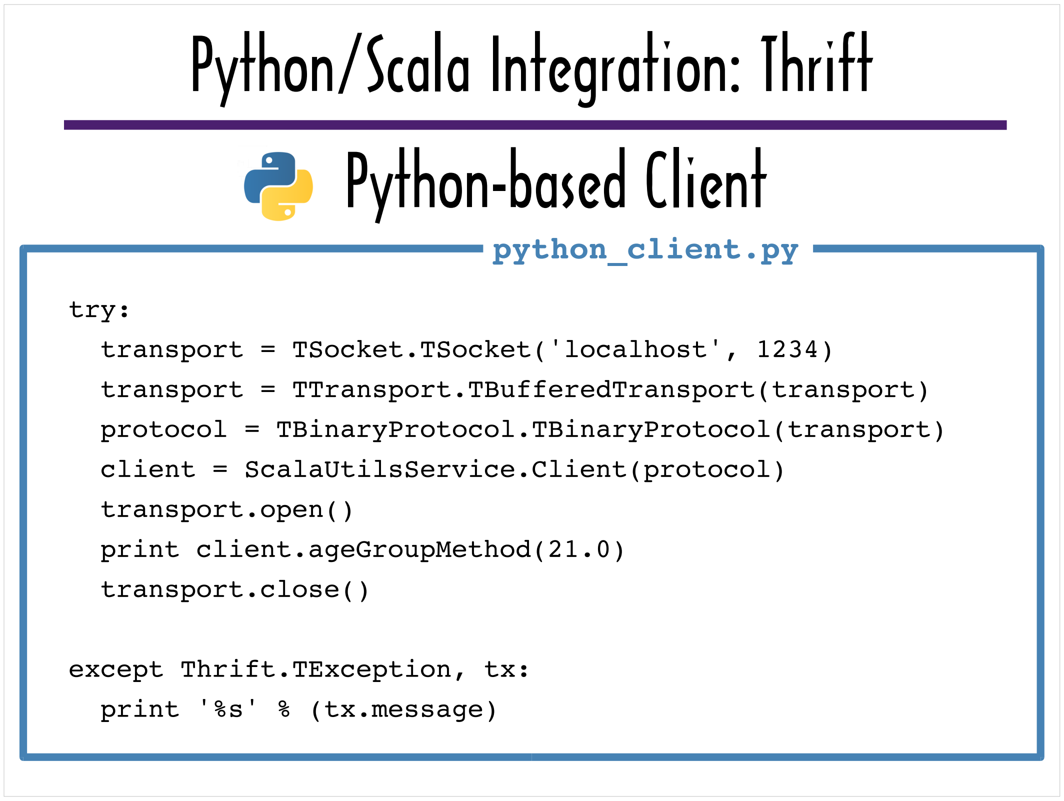
## Python/Scala Integration: Thrift

### Scala-based Server

**scala_utils_server.scala**

```
class ScalaUtilsLogic extends ScalaUtilsService.Iface {
  def ageGroupMethod(age: Double): Int = {
    //Business Logic
  }
}

object ScalaUtilsServer extends Application {
  val serverTransport = new TServerSocket(1234)
  val logic = new ScalaUtilsLogic()
  val processor = new ScalaUtilsService.Processor(logic)
  val plumbing=new Args(serverTransport).processor(processor)
  val server = new TSimpleServer(plumbing)
  server.serve();
}
```

The code highlighted in this slide gives the server access to the communication handlers and class that encapsulates the business logic, and then starts the server.



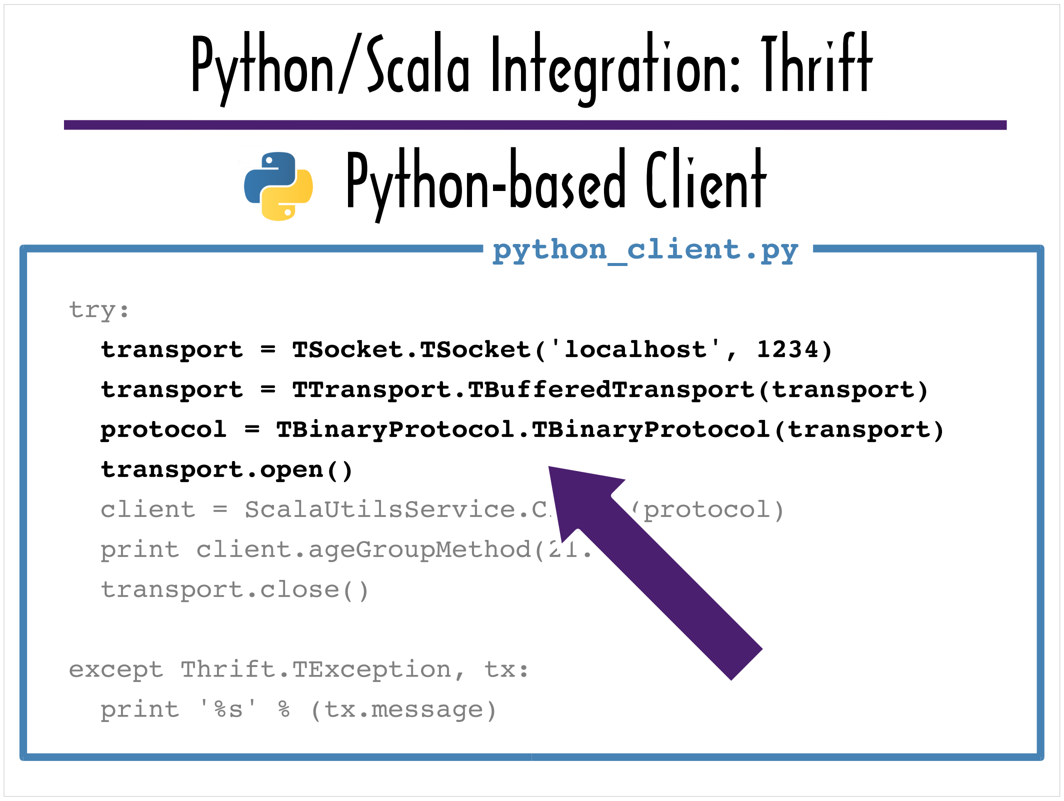
## Python/Scala Integration: Thrift

### Python-based Client

python_client.py

```
try:
  transport = TSocket.TSocket('localhost', 1234)
  transport = TTransport.TBufferedTransport(transport)
  protocol = TBinaryProtocol.TBinaryProtocol(transport)
  client = ScalaUtilsService.Client(protocol)
  transport.open()
  print client.ageGroupMethod(21.0)
  transport.close()

except Thrift.TException, tx:
  print '%s' % (tx.message)
```

Like the Scala-based server, the Python-based client is built on top of Python code generated by Thrift and Thrift core classes.



## Python/Scala Integration: Thrift

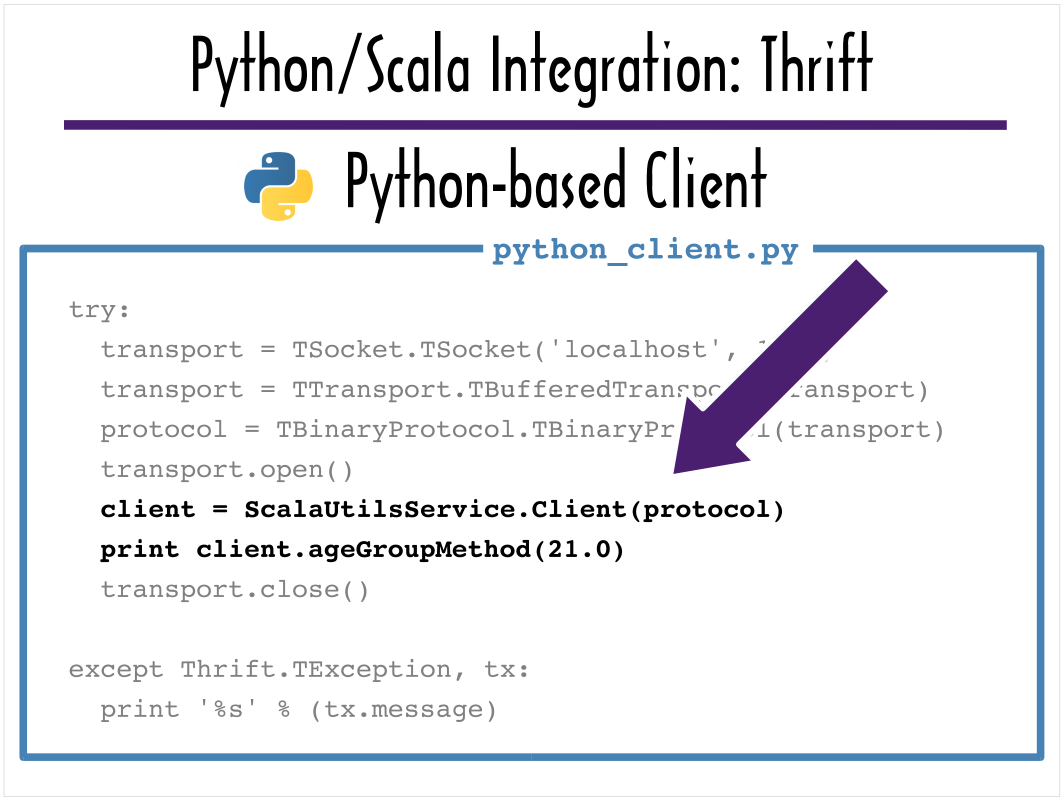
### Python-based Client

python_client.py

```
try:
  transport = TSocket.TSocket('localhost', 1234)
  transport = TTransport.TBufferedTransport(transport)
  protocol = TBinaryProtocol.TBinaryProtocol(transport)
  transport.open()
  client = ScalaUtilsService.C     (protocol)
  print client.ageGroupMethod(21.
  transport.close()

except Thrift.TException, tx:
  print '%s' % (tx.message)
```

The code fragment highlighted here uses the Thrift core classes from the Python library that encapsulate the communication protocol and transport layer.

I used the Thrift's binary protocol, but Thrift ships with additional encoding options, including JSON.



## Python/Scala Integration: Thrift

### Python-based Client

python_client.py

```
try:
  transport = TSocket.TSocket('localhost', 1
  transport = TTransport.TBufferedTranspo      ansport)
  protocol = TBinaryProtocol.TBinaryPr      l(transport)
  transport.open()
  client = ScalaUtilsService.Client(protocol)
  print client.ageGroupMethod(21.0)
  transport.close()

except Thrift.TException, tx:
  print '%s' % (tx.message)
```

ScalaUtilsService, which is highlighted in this slide, is a Thrift-generated Python module and Client is a Thrift-generated class. The Python client can call ageGroupMethod **`ageGroupMethod`** on an instance of ScalaUtilsService.Client.

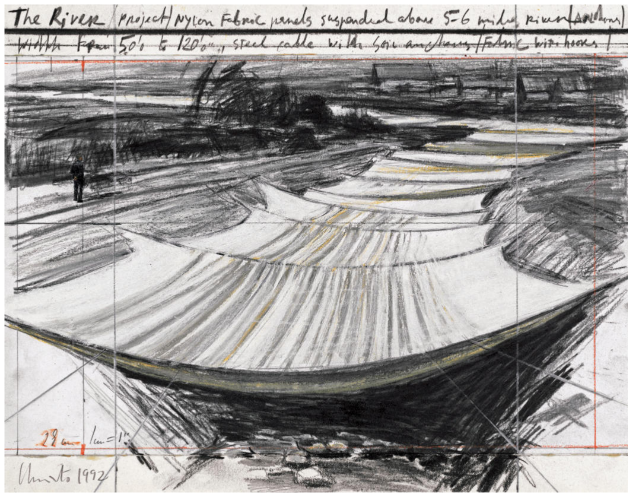


Photo of Christo/Jeanne-Claude sketch by Andre Grossmann:
http://www.christojeanneclaude.net/projects/over-the-river

I hope this presentation has given you some new ideas about how basic communication APIs can be wrapped to facilitate polyglot programming.

I'll wrap up with another image of epic wrapping – wrapping portions of the Arkansas River. This has not been done yet. It's a Christo/Jeanne-Claude sketch for a future project.

Source code for the examples is available at:
https://github.com/A-OK/Snakes-and-Ladders

# POLYGLOT PYTHON: PYTHON/SCALA INTEROP

ANDREA O. K. WRIGHT
Chariot Solutions
https://github.com/A-OK/Snakes-and-Ladders
aok@chariotsolutions.com

Source code for the examples is available at:
https://github.com/A-OK/Snakes-and-Ladders