Python / Java Rosetta Stone

Bob Wilson

September 16, 2016
Purpose / Scope

• The purpose of this guide is to help students entering CS210 from CS110 (now that it is taught in Python) to learn Java
• It presents a side by side comparison of the Python and Java syntax for many common programming constructs
• It is not comprehensive and may not cover subtle but significant semantic differences
Using This Guide

• Don’t just use this guide to “port” Python to Java (You may look like a dork to Java programmers)
  – Python programs are often written “procedurally” (scripts)
  – Java programs should be written “object oriented”
• Although you can write one Java class with a main method and static “helper methods” (functions), that is not the intent of the Java language or the culture of Java programming/programmers
• A good Java program should be decomposed into classes that encapsulate data with operations – not a hierarchy of procedural “functions”
Using This Guide

• Do the top level design of your Java programs with cooperating object oriented classes (e.g. use UML class diagrams – not flow charts or pseudo-code)

• Use this guide to find the corresponding Java syntax for a Python programming construct that you already understand and want to use

• If you wrote good object oriented programs in Python, you just need to learn Java syntax

• If you only wrote procedural programs (scripts) in Python, you need to learn the OOP style plus learn Java syntax (expect you’ll need to do more work)
General Formatting

• Shebang
  #!/usr/bin/env python
• Comments
  # comments for human readers - not code
  statement   # comments to end of line

  """ start of multiple lines of comments
  end of multiple lines of comments """

• Program Statements
  name = expression

• Blocks (Indenting)
  (maybe indented) a statement ending with :
  (indented to next level) starting statement
  (indented to same level) . . .
  (indented to same level) ending statement
  (indented to original or fewer levels)

• Shebang
  Never used or required in Java source code
• Comments
  // comments for human readers – not code
  statement;   // comments to end line

  /* start of multiple lines of comments
  end of multiple lines of comments */

• Program Statements
  (type) name = expression;   // must end with ;

• Blocks (Curly Braces)
  {
    starting statement;
    . . .
    ending statement;
  }   // indenting is used only for readability!!
Key Words / Reserved Words

• Python Key Words
  and  del  from  not  while
  as  elif  global  or  with
  assert  else  if  pass  yield
  break  except  import  print
  class  exec  in  raise
  continue  finally  is  return
  def  for  lambda  try

Notes:
Words in green are not reserved in Java and can be used as identifiers, etc.

There are also some type and constant names:
  int, float, True, False, None, etc.
  that correspond to reserved words in Java
  maybe with different spelling or capitalization:
  int, float, true, false, null, etc.

• Java Reserved Words
  abstract  default  goto*  package  this
  assert  do  if  private  throw
  boolean  double  implements  protected  throws
  break  else  import  public  transient
  byte  enum  instanceof  return  true
  case  extends  int  short  try
  catch  false  interface  static  void
  char  final  long  strictfp  volatile
  class  finally  native  super  while
  const*  float  new  switch
  continue  for  null  synchronized
* Are reserved words, but are not used.

Notes:
Words in black have generally the same semantics in Java as they do in Python.
If you have been using any of the red words in Python, you will need to avoid using them in Java.
Primitive Data Types

• Numeric Data Types
  
  int  | Natural Numbers (Integers)
  long | Large Natural Numbers
  float | Real Numbers (Decimal)
  complex | Complex Numbers (R + I * j)

• Other Data Types
  
  boolean | Logical “True” or “False” values
  class | Any defined class as a type
  string | An array of characters

• Numeric Data Types
  
  byte  | 8 Bit Numbers
  char  | 16 Bit Unicode Characters
  short | 16 Bit Numbers
  int   | 32 Bit Numbers
  long  | 64 Bit Numbers
  float | Real Numbers (Decimal)
  double | Larger/Smaller Real Numbers

• Other Data Types
  
  boolean | Logical “true” or “false” values
  Class  | Any defined class as a type
  String | A somewhat special class
  Interface | Any defined interface as a type
## Primitive Data Constants

### Type int / long

- **Decimal**: 123  \(\#\ 123_{10}\)
- **Octal**: 0123  \(\#\ 83_{10}\)
- **Hex**: 0x123  \(\#\ 291_{10}\)
- **Binary**: 0b101  \(\#\ 5_{10}\)
- **long**: 1234567890123456789L

### Type int / long

- **Decimal**: 123  \(\#\ 123_{10}\)
- **Octal**: 0123  \(\#\ 83_{10}\)
- **Hex**: 0x123  \(\#\ 291_{10}\)
- **Binary**: 0b101  \(\#\ 5_{10}\)
- **long**: 1234567890123456789L

Note: In Java, long has a smaller maximum number of digits than in Python

### Type float

- **float**: 123.0  \(\# 123.0\)
- **float**: 1.23e308  // 1.23 \(\times\ 10^{308}\)
- **float**: 1.23e-308  // 1.23 \(\times\ 10^{-308}\)

### Type float / double

- **float**: 123.0f  // 123.0
- **float**: 1.23e38f  // 1.23 \(\times\ 10^{38}\)
- **float**: 1.23e-38f  // 1.23 \(\times\ 10^{-38}\)
- **double**: 1.23e308  // 1.23 \(\times\ 10^{308}\)
- **double**: 1.23e-308  // 1.23 \(\times\ 10^{-308}\)

Note: Type double is default for real in Java

Conversion needed to get desired type:

\[ i = \text{int}(123.4) \quad \# i = 123 \]
\[ f = \text{float}(i) \quad \# f = 123.0 \]

Casting needed for narrowing conversions:

\[ \text{float} f = (\text{float}) 123.4; \quad // \text{double to float} \]
\[ \text{int} i = (\text{int}) f; \quad // \text{float to int} \ 123 \]
Variables

• Declarations
All variables are “reference” types. Variables do not need to be declared. A variable is created by initializing it and its type is determined by the type of the value assigned:

\[ i = 10 \quad \text{# i is an int} \]

Its type can be changed later:

\[ i = 10.5 \quad \text{# i is a float now} \]

A variable can be deleted (undefined):

\[ \text{del } i \]

Using i in an expression is invalid now unless it is initialized again.

• Declarations
There are primitive and reference variables. All variables must be declared before use. A variable is created by declaring it with its data type and optionally initializing it.

A **primitive** variable is of a built in data type:

\[ \text{int } i = 10; \quad \text{// i is an int} \]

Its type can not be changed later:

\[ i = 10.5; \quad \text{// compilation error} \]

A **reference** variable is of a user defined type based on a class or is reference to an array:

\[ \text{String myString} = \text{“Hello”}; \]
\[ \text{int [ ] myNumbers} = \text{new int[10]}; \]

A variable can not be deleted (undefined).
Operators

• **Arithmetic Operators**
  +  add, e.g. 4 + 2 is 6
  -  subtract, e.g. 4 – 2 is 2
  *  multiply, e.g. 4 * 2 is 8
  /  divide, e.g. 4 / 2 is 2 (dividend)
  %  modulo, e.g. 4 % 2 is 0 (remainder)
  ** exponentiation, e.g. 4 ** 2 is 16

Note: ++ and -- are NOT Python operators

• **Logical Operators**
  and  and (between boolean values)
  or  or (between boolean values)
  not  not (of a boolean value)
  &  Bitwise and (between int values)
  |  Bitwise or (between int values)
  ^  Bitwise exclusive or (between int values)
  <<  Bitwise Left Shift (of an int value)
  >>  Bitwise Right Shift (of an int value)

• **Arithmetic Operators**
  +  add, e.g. 4 + 2 is 6
  -  subtract, e.g. 4 – 2 is 2
  *  multiply, e.g. 4 * 2 is 8
  /  divide, e.g. 4 / 2 is 2 (dividend)
  %  modulo, e.g. 4 % 2 is 0 (remainder)

Note: ** is NOT a Java operator
  ++  pre/post increment by one
  --  pre/post decrement by one

• **Logical Operators**
  &&  and (between boolean values)
  ||  or (between boolean values)
  !  not (of a boolean value)
  &  Bitwise and (between int values)
  |  Bitwise or (between int values)
  ^  Bitwise exclusive or (between int values)
  <<  Bitwise Left Shift (of an int value)
  >>  Bitwise Right Shift (of an int value)
Expressions

• **Operator Precedence**
  Same in Python and Java (Algebraic)
  Override precedence with parentheses ( )

• **Casting / Conversions**
  **Numeric Casting/Conversions**
  Automatic widening type conversions,
  e.g. 1 + 3.0 results in a float 4.0
  Functions required for narrowing conversions,
  e.g. 1 + int(3.0) results in an int 4
  **Non-numeric Conversions**
  Need to use conversion functions,
  e.g int("string of digits") which
  raises an Error for non-digit characters

• **Operator Precedence**
  Same in Python and Java (Algebraic)
  Override precedence with parentheses ( )

• **Casting / Conversions**
  **Numeric Casting/Conversions**
  Automatic widening type conversions,
  e.g. 1 + 3.0 results in a double 4.0
  Casting required for narrowing conversions,
  e.g. 1 + (int) 3.0 results in an int 4
  **Non-numeric Conversions**
  Need to use wrapper class static methods,
  e.g Integer.parseInt("string of digits") which
  throws an Exception for non-digit characters
Stand-alone Functions / Methods

- Function Definition
  ```python
def function (parameters):
    statements
    return value
  ```

- Invoking a Function
  ```python
  # no context of an object or class is required
  returnValue = function( . . .)
  ```
  ```python
e.g.
  length = len(myString)
  ```
  ```python
  // using a function defined in the library
  returnValue = packageName.function(. . .)
  ```
  ```python
e.g.
  import math  # library package name
  c = math.sqrt(2.0)  # 1.414…
  ```

- No Equivalent in Java
  A function can only be defined as a method within the context of a class or an interface. See Classes and Java 8 Lambda Expressions

- Invoking a Method
  ```python
  // the context of an object or class is required
  // instance method (non static)
  type returnValue = object.method( . . .);
  ```
  ```python
e.g.
  int length = myString.length();
  ```
  ```python
  // static method (defined in a class, e.g. Math)
  type returnValue = Class.method( . . .);
  ```
  ```python
e.g.
  import math  # library package name
  double root = Math.sqrt(2.0);  // 1.414…
  ```
String Data Type

- **Strings**
  myString = "Hello World"
  myString = 'Hello World'
  myString = """Hello World""
  Note: "\n" is end of line in a string

- **String Functions**
  \[ n = \text{len}(\text{myString}) \quad \# n = 11 \]
  \[ c = \text{myString}[0] \quad \# c = "H" \]
  \[ s = \text{myString}[0 : 2] \quad \# s = "He" \]
  \[ s = \text{myString}.upper() \quad \# s = "HELLO" \]

- **String Operations**
  \[ s = \text{myString} + "!" \quad \# \text{Concatenation} \]
  \[ s = \text{myString} + \text{str}(42) \quad \# \text{HelloWorld42} \]
  myString == "Hello World" \quad \# True

- **String Class / char**
  String myString = "Hello World";
  char c = 'a'; \quad // 'a' = char constant for letter a
  Note: ‘\n’ is end of line in a char
  Note: “\n” is end of line in a String

- **String Methods / char**
  \[ \text{int} n = \text{myString}.\text{length}(); \quad \# n = 11 \]
  \[ \text{char} c = \text{myString}.\text{charAt}(0); \quad \# c = 'H' \]
  String s = myString.substring(0, 2); \quad // s = "He"
  s = myString.toUpperCase(); \quad // "HELLO"

- **String Operations**
  \[ s = \text{myString} + "!"; \quad \# \text{Concatenation} \]
  \[ s = \text{myString} + 42; \quad \# \text{HelloWorld42} \]
  myString.equals("Hello World") \quad \# true
Multi-valued Data Types

• Lists
  Python lists are a dynamic data structure.
  Java arrays are a FIXED data structure.

  ```python
  anEmptyList = [ ]        # type unspecified
  myList = ["you", "me", "him", "her"]
  length = len(myList)     # 4
  myList[0]                # "you"

  myList[3]                # "her"
  myList[0] = "thee"      # update an element
  
  List methods in Python:
  myList.sort()          # sort the elements
  myList.reverse()       # reverse the elements
  myNums.append(5)       # add an element
  myNums.remove(3)       # remove one
  ```

• Arrays
  Syntax for a Java array looks like a Python list, BUT THE SEMANTICS ARE DIFFERENT!

  ```java
  int [ ] anEmptyArray= new int[10];   // type int
  String [ ] myList = {"you", "me", "him", "her"};
  int length = myList.length;         // 4
  myList[0]                           // "you"

  myList[3]                           // "her"
  myList[0] = "thee";                 // update an element
  
  There are NO methods for a Java array
  No equivalent with Java arrays
  No equivalent with Java arrays.
  No equivalent with Java arrays.
  
  Length of a Java array can't be changed.
  Must use Java Collections class ArrayList<T>.
  We will cover collection classes in CS210.
  ```
Multi-valued Data Types

• **Tuples**

```python
person = ("Diana", 32, "New York")
person[0]  # "Diana"
person[1]  # 32
```

```python
person[0] = "Amy"  # not allowed
```

```python
person = person + person (concatenate)
```

```python
Person[3]  # "Diana" (again)
```

• **Dictionaries**

```python
words = { }  # empty
words["Hello"] = "Bonjour"
words["Goodbye"] = "Adieu"
```

```python
words["Hello"]  # "Bonjour"
words["Yes"]  # raises an Error
```

```
KeyError: “Yes”
```

• **No Equivalent Type in Java**

A Java object can be used as a specific “tuple”. Define a class with the needed combo of types.

- Attributes of the class are the items.
- Setter and getter methods allow access - not [ ]

**BUT:**

- We MAY allow updating of item values.
- We can NOT concatenate objects (except String)

(See Classes)

• **No Equivalent Type in Java**

Must use a Java Collections map class e.g. HashMap<K,V> or TreeMap<K,V>.

We will cover these classes in CS210.
Input / Output

• Input (Command Line)
  python script.py tokens separated by spaces

• Program Arguments
  Note: No main function header is required
  import sys                   # but import is required
  n = len(sys.argv)            # n = 5
  firstArg = sys.argv[0]       # “script.py”
  ...                         
  lastArg = sys.argv[4]        # “spaces”

  # if second token should be an integer,  
  n = int(sys.argv[1])

  # if last token should be a float,      
  f = float(sys.argv[4])

• Input (Command Line)
  java classname tokens separated by spaces

• Main Method Arguments
  public static void main (String[ ] args)  
  {  
      int n = args.length;       // n = 4
      String firstArg = args[0]; // “tokens”
      ...                         
      String lastArg = args[3];  // “spaces”

      // if first token should be an integer, 
      int n = Integer.parseInt(arg[0]);

      // if last token should be a double,  
      double d = Double.parseDouble(arg[3]);
  }
Input / Output

• **Typed Outputs to User**
  ```java
  System.out.println("Text String");
  ```

• **User Prompt/Response**
  ```java
  import java.util.Scanner;
  Scanner keyboard = new Scanner(System.in);
  String s = keyboard.next(); // token
  int n = keyboard.nextInt(); // integer
  float f = keyboard.nextFloat(); // real
  double d = keyboard.nextDouble(); // double
  boolean b = keyboard.nextBoolean(); // boolean
  ```

• **Typed Outputs to User**
  ```java
  print ("Text String")
  ```

• **User Prompt/Response**
  ```java
  s = input("Prompt")                       // token
  n = int(input("Prompt:"))               // integer
  f = float(input("Prompt:"))             // real
  ```
Flow of Control Statements

- **If / Else**
  
  ```python
  if boolean expression:
      statement1 or block1
  else:     # optional
      statement2 or block2
  
  May nest “if/else” inside “if” or “else”
  Python “elif” must be “else if” in Java
  ```

- **Conditional Expression Evaluation**
  
  ```python
  ((False expr, True expr) [condition])
  ```

- **Conditional Boolean Operators**
  
  ```plaintext
  ==  equal
  !=  not equal
  >   greater than
  <   less than
  ```
Flow of Control Statements

- **For**
  
  for i in range(0, 10, 1):
      statement or block using i

  for item in items:
      # items is a list
      statement or block using item

- **While**

  while boolean expression:
      statement or block for body of loop

- **For**

  for (int i = 0; i < 10; i++)
      single statement; or {block}

  // sometimes referred to as a “for-each” loop
  for (type item : items)     // items is an array
      single statement; or {block}

- **While**

  while (boolean expression)
      single statement; or {block}

- **Do . . . while**

  do       // always executes body once
      single statement; or {block}
  while (boolean expression);
Classes

• Class Definition
  class ClassName:
   attributes and methods

• Public Attribute
  name (optional = value)

• Private Attribute
  __name (optional = value)
  Note: A programmer convention only
  Access IS NOT prevented by interpreter

• Conventional Word “self”
  Used to refer to your own object in Python
  You may use another word, but “self” is the commonly accepted convention.

• Class Definition
  public class Classname
  {
    attributes and methods
  } // end of class definition

• Public Attribute
  public (static) type name (optional = value);

• Private Attribute
  private (static) type name (optional = value);
  Note: Access IS prevented by compiler

• Reserved Word “this”
  Used similarly to “self” in Python
  You must use the reserved word “this”.
  Not required in as many places in the code, e.g. not needed in method parameter lists.
Classes

- **Constructor Method**
  ```python
  def __init__(self, parameter):
      self.parameter = parameter
  ```

- **Public Method**
  ```python
  def name(self, parameters):
      statements
  ```

- **Private Method**
  ```python
  def __name__(self, parameters):
      statements
  Note: A programmer convention only
  Access IS NOT prevented by interpreter
  ```

- **Constructor Method**
  ```java
  public ClassName (parameter) {
      this.parameter = parameter;
  } // end of method
  ```

- **Public Method**
  ```java
  public type name (parameters) {
      statements;
  } // end of method
  ```

- **Private Method**
  ```java
  private type name (parameters) {
      statements;
  } // end of method
  Note: Access IS prevented by compiler
  ```
Classes

• Method Return Value
  
  def name (self, parameters):
      return expression

• Method Overloading
  
  def name (self, param = None):
      if param is None:
          1st version of statements
      else:
          2nd version of statements

• Method Return value
  
  public type name (parameters)
  {
      return expression of type;
  }  // end of method

• Method Overloading
  
  public type name ( )  // no parameter
  {
      1st version of statements;
  }  // end of first “name” method

  public type name (type param)
  {
      2nd version of statements;
  }  // end of second “name” method
Python “Magic” Methods

• Magic Methods
  ```python
  __str__(self) # representation
  __cmp__(self, other) # compare objects
  (Supports operator overloading for >, <, etc.)
  __add__(self, other) # and sub, mul, div, etc
  (Supports operator overloading for +, -, *, /, etc.)
  __eq__(self, other) # check equality
  __iter__(self) # returns an iterator
  (Supports “for item in items” type of loop)
  __del__(self) # clean up
  ```

• Java Equivalents
  ```java
  public String toString() // representation
  public int compareTo(that) // compare objects
  (Supports implementing Comparable interface)
  public boolean equals(that) // check equality
  public Iterator<T> iterator() // returns an interator
  (Supports “for (type item : items)” for-each loop and implementing Iterable<T> interface)
  protected void finalize() // clean up
  ```

Note: Java operator overloading is not supported.
Creating / Deleting Objects

- **Instantiating an Object**
  
  ```
  myObject = ClassName(. . .)
  # … are values for constructor’s parameters
  ```

- **Creating an Alias**
  
  ```
  yourObject = myObject
  # … both variables refer to the same object
  ```

- **Deleting an Object**
  
  ```
  myObject = None      # deletes object
  # (if there is no alias)
  ```

- **Instantiating an Object**
  
  ```
  Classname myObject = new ClassName( . . . );
  // … are values for constructor’s parameters
  ```

- **Creating an Alias**
  
  ```
  ClassName yourObject = myObject;
  # … both variables refer to the same object
  ```

- **Deleting an Object**
  
  ```
  myObject = null;  // deletes object
  # (if there is no alias)
  ```
Inheritance / Interfaces

• Inheritance
  # OO Concept: A Cat is an Animal
class Cat(Animal):
    attributes and methods

• Multiple Inheritance
class ClassName(Class1, Class2, …):
    attributes and methods

• Inheritance
  // OO Concept: A Cat is an Animal
class Cat extends Animal
{
    attributes and methods
} // end of class

• No Multiple Inheritance
  Java doesn’t support more than one parent class

• Interfaces
  Java supports implementing multiple interfaces
public class ClassName implements Int1, Int2, …
{
} // end of class
Inheritance / Interfaces

- Polymorphism

```python
class Pet:  # abstract parent class
def makeSound(self):
    raise NameOfError("text")

class Dog(Pet):  # concrete child class
    def makeSound(self):
        print "Woof Woof"

class Cat(Pet):  # concrete child class
    def makeSound(self):
        print "Meow"

spot = Dog()
spot.makeSound()  # Woof Woof
fluffy = Cat()
fluffy.makeSound()  # Meow
```

# Attempt to create/use an abstract class
fubar = Pet()
fubar.makeSound()  # raises an Error
# at run time

- Polymorphism

In Java, a reference to any object may be saved as a reference to the type of a parent class or of any implemented interface:

If Cat class and Dog class extend Pet class, we can do these “widening” conversions:

```java
Dog d = new Dog();
Pet p = d;  // our Pet is a Dog
p = new Cat();  // and is now a Cat
```

And call any Pet method on variable `p`:

```java
p.anyPetMethod(...);  // on Dog/Cat
```

If a method parameter needs to be a Pet, we can pass a Dog or a Cat object to it:

```java
public void methodName(Pet p) {...}
methodName(d);  // pass it a Dog
methodName(new Cat());  // or Cat
```

If Pet is an abstract class, we can’t create a Pet object (causes a compilation error):

```java
Pet p = new Pet();  // compile error
```
Inheritance / Interfaces

• Polymorphism

If a method definition requires returning a reference to a class or interface, it may return a reference to an object of the class, a child class, or an implementing class.

If Pet class implements Comparable<T>, Dog and Cat class also implement it. If we invoke a method with a return value of type Comparable<T>:

```java
Comparable<T> c = methodName( . . . );
```

It can return a Dog or a Cat object:
```java
public Comparable<T> methodName( . . . )
{
    if (some boolean expression)
        return new Dog();
    else
        return new Cat();
}
```
Errors / Exceptions

• Errors
Because Python code is interpreted, many syntax errors are detected only at run time.
>>> while True print ‘Hello World’ # no:
while True print ‘Hello World’

^ SyntaxError: invalid syntax

To raise an error in your code:
if something bad would happen:
    raise NameOfError("text")

To handle a run time error - not syntax error
try:
    statements that could raise an error
except NameOfError:
    statements to recover from the error
else:
    statements executed if no error raised

• Exceptions
In Java, all syntax errors are caught during compilation and before run time.

Exceptions occur during runtime only if:
1. JVM can’t execute, e.g. int divide by 0
2. Code throws an exception object

To throw an exception in your code:
if (something bad would happen)
    throw new NameOfException("text");

To handle an exception in your code:
try {
    statements that may throw an exception
} catch (NameOfException e) {
    statements to recover from the exception
} finally {
    statements to execute regardless
}
Lambda Expressions

```java
import math
f = lambda x, y : math.sqrt(x * x + y * y)

c = f(3, 4)                 # c = 5.0
```

Lambda Expressions

```java
public class LambdaTest {
    interface MyMath {      // a functional interface
        int operation (int a, int b); // only 1 method
    }

    public int operate(int a, int b, MyMath math) {
        return math.operation (a, b);
    }

    public static void main(String[] args) {
        // alternative definitions for operation to add
        MyMath add = (int a, int b) -> a + b;   // or
        MyMath add = (a, b) -> a + b;           // or
        MyMath add = (a, b) -> {return a + b; }

        LambdaTest tester = new LambdaTest();
        int n = tester.operate(2, 5, add)    // n = 7
    }
}
```