Java for Python Programmers

- Comparison of Python and Java Constructs
- Reading: L&C, App B

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 float

float	123.0	# 123.0
float	1.23e308	// 1.23 x 10^{308}
float	1.23e-308	// 1.23 x 10^{-308}

Conversion needed to get desired type:

```
i = int(123.4)          # i = 123
f = float(i)            # f = 123.0
```

- Type int / long

Decimal	123	# 123_{10}	
Octal	0123	# 83_{10}	
Hex	0x123	# 291_{10}	
Binary	0b101	# 5_{10}	(Java 8)
long	1234567890123456789L		

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

- Type float / double

float	123.0f	// 123.0
float	1.23e38f	// 1.23 x 10^{38}
float	1.23e-38f	// 1.23 x 10^{-38}
double	1.23e308	// 1.23 x 10^{308}
double	1.23e-308	// 1.23 x 10^{-308}

Note: Type double is default for real in Java

Casting needed for narrowing conversions:

```
float f = (float) 123.4;   // double to float
int i = (int) f;           // 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            # i is an int
```

Its type can be changed later:

```
i = 10.5          # i is a float now
```

A variable can be deleted (undefined):

```
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

```
int i = 10;              // i is an int
```

Its type can not be changed later:

```
i = 10.5;                // compilation error
```

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

```
String myString = “Hello”;
int [ ] myNumbers = 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

```
def function (parameters):
    statements
    return value
```

- No Equivalent in Java

A function can only be defined as a method within the context of a class or an interface. (See Classes)

- Invoking a Function

```
# no context of an object or class is required


returnValue = function( . . .)
e.g.
length = len(myString)

// using a function defined in the library
returnValue = packageName.function(. . .)
e.g.
import math            # library package name
c = math.sqrt(2.0)     # 1.414…
```

- Invoking a Method

```
// the context of an object or class is required

// instance method (non static)
type returnValue = object.method( . . .);
e.g.
int length = myString.length();

// static method (defined in a class, e.g. Math)
type returnValue = Class.method( . . .);
e.g.
// Note: Math class is automatically imported
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 = len(myString)   # n = 11
c = myString[0]   # c = “H”
s = myString[0 : 2]   # s = “He”
s = myString.upper()   # s = “HELLO”

- String Operations

s = myString + “!”   # Concatenation
s = myString + str(42)   # HelloWorld42
myString == “Hello World”   # True

- String Class / char

String myString = “Hello World”**;**
char c = ‘a’**;**   // ‘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

int n = myString**.**length()**;**   // n = 11
char c = myString**.**charAt(0)**;**   // c = ‘H’
String s = myString**.**substring(0, 2)**;** // s = “He”
s = myString**.**toUpperCase()**;**   // “HELLO”

- String Operations

s = myString + “!”**;**   **//** Concatenation
s = myString + 42**;**   // HelloWorld42
myString**.**equals(”Hello World”)   // true

Multi-valued Data Types

- Lists

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

```
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!

```
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.
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

```
person = (“Diana”, 32, “New York”)
person[0]                # “Diana”
person[1]                # 32
person[2]                # “New York”

person[0] = “Amy”   # not allowed
person = person + person (concatenate)
Person[3]               # “Diana” (again)
```

- 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:
Java can allow updating of item values.
We can NOT concatenate objects (except String)
(See Classes)

- Dictionaries

```
words = { }                    # empty
words[“Hello”] = “Bonjour”
words[“Goodbye”] = “Adieu”

words[“Hello”]              # “Bonjour”
words[“Yes”]                # raises an Error
KeyError: “Yes”
```

- 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.

Flow of Control Statements

- If / Else

if boolean expression**:**
    statement1 or block1
else**:**   # optional
    statement2 or block2

May nest “if” or “else” inside “if” or “else”

Conditional Expression Evaluation

Not supported in Python

Conditional Boolean Operators

==	equal
!=	not equal
>	greater than
<	less than

- If / Else

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

boolean expression **?** true expr **:** false expr

Conditional Boolean Operators

==	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
```

Note: Loops may be nested in Python and Java

- 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);
```

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

```
print ("Text String")              # with EOL
print ("text String", end = "")  # no EOL
```

- User Prompt/Response

```
s = input("Prompt")                   // token
n = int(input("Prompt:"))             // integer
f = float(input("Prompt:"))           // real
```

- Typed Outputs to User

```
System.out.println("Text String"); // with EOL
System.out.print("Text String");    // no EOL
```

- User Prompt/Response

```
import java.util.Scanner;    // at beginning of file
. . .
Scanner keyboard = new Scanner(System.in);
System.out.println("Prompt:");
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
```

File Input: Example

```
import java.util.Scanner;
import java.io.*;

public class FileDisplay
{
  public static void main (String [] args)
                              throws IOException
  {
    Scanner scan = new Scanner(System.in);
    System.out.println("Enter name of file to display");
    File file = new File(scan.nextLine()); // open file

    Scanner fileScan = new Scanner (file);
    while (fileScan.hasNext())
      System.out.println(fileScan.nextLine());
  }
}
```

File Output: Example

```
import java.util.Scanner;
import java.io.*;

public class FileWrite
{
  public static void main (String [] args)
                                          throws IOException
  {
    Scanner scan = new Scanner(System.in);
    System.out.println("Enter name of file to display");
    File file = new File(scan.nextLine()); // open file

    PrintStream out = new PrintStream(file);
    while (scan.hasNext()) {
      String line = scan.nextLine(); // read keyboard
      out.println(line);             // write file
    }
    out.close();                     // close file
  }
}
```

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
}
```