Differences Between Java and Python

- Java
  - compiled
- Python
  - interpreted

Differences Between Java and Python

- Java
  - compiled
- Python
  - interpreted

every statement
in a source file
is executed
immediately

Differences Between Java and Python

- Java
  - compiled
  - static typing

- Python
  - interpreted
  - dynamic typing

Differences Between Java and Python

- Java
  - compiled
  - static typing
  - blocks delimited with { }
- Python
  - interpreted
  - dynamic typing
  - blocks delimited by indentation

Differences Between Java and Python

- Java
  - compiled
  - static typing
  - blocks delimited with { }
  - more verbose
    - variable declarations
    - each public class requires a separate file
    - exception propagation must be declared
- Python
  - interpreted
  - dynamic typing
  - blocks delimited by indentation
  - less verbose
    - just use the variable
    - multiple classes can be defined in one file
    - exceptions propagate upwards automatically

Differences Between Java and Python

- Java
  - compiled
  - static typing
  - blocks delimited with { }
  - more verbose
    - variable declarations
    - each public class requires a separate file
    - exception propagation must be declared
  - lists and hash tables provided by libraries

- Python
  - interpreted
  - dynamic typing
  - blocks delimited by indentation
  - less verbose
    - just use the variable
    - multiple classes can be defined in one file
    - exceptions propagate upwards automatically
  - lists and hash tables are native types

Similarities Between Java and Python

- Good cross-platform support
- (Almost) everything is an object
- Compile down to bytecode for a virtual machine
- Strongly typed (but Python variables change type depending on content)
- Both use garbage-collected automatic memory managment

Variable Typing

- Java

```
boolean x = true;

int x = 1;

float x = 2.5;

String x = new String(“s”);

List x;

Hashtable x;

Complex x;
```

- Python

```
x = True

x = 1

x = 2.5

x = 's'

x = [1, '2', [3.5, 4], 5]

x = {}

x = 1 + 2j
```

Python variables are really just pointers to objects

lists can contain arbitrary types

Statement Terminators

- Java

```java
// semicolon terminates
// statements

f = 2.5; i = 1; s1 = s2;
```

- Python

```python
# end-of-line ends statement
f = 2.5
i = 1
s1 = s2

# use backslash or unclosed
#   parens to continue
f = sin(4.6*cos(y)) \
   + tan(z)
f = (1 + 2 + 3 + 4.6
   + tan(z))

# semicolon allowed but
# discouraged
f = 2.5; i = 1
```

Block Scoping

- Java

```
// using braces:


if (x < y) { ... } else
{ ... }



if (y.equals(z))
{
   conditional1();
   conditional2();
}
common();
```

- Python

```
# using indentation:


if x < y:
   ...
else:
   ...


if y == z:
   conditional1()
   conditional2()
common()
```

indentation matters!

Library Import

- Java

```java
import library;
import library.*;
```

- Python

```python
# import a module into a new
# namespace
import library
import library as alias

# import specific object(s)
# from a module
from library import obj, obj2

# import entire module into
# local namespace
from library import *
```

Library Import

- Java

```
import library;
import library.*;
```

- Python

```
# import a module into a new
# namespace
import library
import library as alias

# import specific object(s)
# from a module
from library import obj, obj2

# import entire module into
# local namespace
from library import *
```

obj and *obj2* can be accessed as if they had been defined in the current file

Library Import

- Java

```java
import library;
import library.*;
```

Dangerous! Public symbols in *library* will replace any local symbols with the same name

- Python

```python
# import a module into a new
# namespace
import library
import library as alias

# import specific object(s)
# from a module
from library import obj, obj2

# import entire module into
# local namespace
from library import *
```

Equality Tests

- Java

```
// object identity:
//    ==

// equal values:
//    .equals()


  if (x == y) {
    System.out.print(“Same”);
  }
  if (x.equals(y)) {
    System.out.print(“Equal”);
  }
```

- Python

```
# object identity:
#    is

# equal values:
#    ==


  if x is y:
    print “Same”

  if x == y:
    print “Equal”
```

in Python3, **print** is a function: print(“Same”)

Special Pointers

- Java

```
// invalid/“Null” pointer:
//    null

// Current object:
//    this


    this.value = null;
```

- Python

```
# invalid/“Null” pointer:
#    None

# Current object:
#    self


    self.value = None
```

“self” is a convention, **not** a keyword!

Function Declaration

- Java

```
rettype funcname
  ( argtype argname, ... )
{
   rettype result = X;
   // body
   return result;
}
```

- Python

```
def funcname( argname, ... ):
   result = X
   //body
   return result
```

Function Declaration

- Java

```
rettype funcname
  ( argtype argname, ... )
{
   rettype result = X;
   // body
   return result;
}
```

- Python

```
def funcname( argname, ... ):
   result = X
   //body
   return result
```

def statements are executed immediately, generating a function object and binding it to a name in the current module

Class Declaration

- Java

```
class cl extends X {
   type data = defvalue;

   type func(type N) {
      return N * data;
   }
}
```

- Python

```
class cl(X):
   data = defvalue

   def func(self, N):
      return N * self.data
```

Python does not have an implicit object pointer on class method declarations

Class Declaration

- Java

```
class cl extends X {
   type data = defvalue;

   type func(type N) {
      return N * data;
   }

   static int fact(int N) {
      if (N < 2) return 1;
      return N * fact(N-1);
   }
}
```

- Python

```
class cl(X):
   data = defvalue

   def func(self, N):
      return N * self.data

   @staticmethod
   def fact(N):
      if N < 2:
         return 1
      return N * cl.fact(N-1)
```

Accessing Class Members

- Java

```java
cl foo = new cl();

type d1 = foo.data;
type d2 = foo.func(X);
int v = foo.fact(5);
```

- Python

```python
foo = cl()

d1 = foo.data
d2 = foo.func(X)
v = foo.fact(5)
```

Exception Handling

- Java

```
class E extends Exception;

void foo() throws E {
   throw new E();
}

void bar() {
   try {
      foo();
   } catch (E err) {
      System.out.print(“Err”);
   } finally {
      System.out.print(“Always”);
   return;
}
```

- Python

```
class MyErr(RuntimeError):
   pass     # do nothing

def foo():
   raise MyErr(“msg”)

def bar():
   try:
      foo()
   except MyErr as err:
      print “Err: ”, err
   else:
      print “Success”
   finally:
      print “Always”
```

Lists in Python

```
# instantiate an empty list
l1 = []
# instantiate list with heterogenous values
l2 = [1,'foo',3.5]
# instatiate list of 100 references to an item
l3 = 100*['item']

# print sub-list, from index i to (but not including) index j
print l2[1:2]            #==> ['foo']
print l2[1:3]            #==> ['foo', 3.5]

# negative indices count from end of list
print l2[-2:]            #==> ['foo', 3.5]
```

this is a valid Python 2 program!

Tuples in Python

```
# tuples are immutable lists

# instantiate a tuple
t1 = (1, 2, 3)
# optionally leave out the parentheses
t2 = 1, 2, 3

print t1, t2              #==> (1, 2, 3) (1, 2, 3)

# commonly used to return multiple values:
x = 0.125
num, denom = x.as_integer_ratio()
print num                 #==> 1
print denom               #==> 8
```

Tuples in Python

```
# tuples are immutable lists

# instantiate a tuple
t1 = (1, 2, 3)
# optionally leave out the parentheses
t2 = 1, 2, 3

print t1, t2             #==> (1, 2, 3) (1, 2, 3)

# commonly used to return multiple values:
x = 0.125
num, denom = x.as_integer_ratio()
print num                #==> 1
print denom              #==> 8
```

tuple automatically unpacked into multiple variables

List Comprehensions

```
# a very compact way to generate lists
even_squares = [n**2 for n in range(1000) if n%2 == 0]
print even_squares  ==> [0, 4, 16, 36, 64, ..., 992016, 996004]

# for expressions can be nested to generate tuples
cards=[(rank,suit) for rank in [2,3,4,5,6,7,8,9,10,'J','Q','K','A']
         for suit in ['C','D','H','S']]
print cards     ==> [(2,'C'), (2,'D'), ..., ('A','H'), ('A','S')]


# general syntax:
#   [ expression for var in iterable if condition ]
# "for var in iterable" may be repeated with multiple variables and
# iterators; "if condition" is optional
```

Hash Maps in Python

```
# instantiate an empty hash map (called a “dictionary”)
ht = {}

# insert values
ht[5] = [1,2,3,4,5]
ht['foo'] = 'Yes'
print ht                ==> {'foo': 'Yes', 5: [1, 2, 3, 4, 5]}

# retrieve a value
print ht[5]             ==> [1, 2, 3, 4, 5]
print ht['foo']         ==> Yes

# remove lookup key
del ht['foo']

# attempting to access removed key generates an error:
print ht['foo']         ==> KeyError: 'foo'
```

Sets in Python

```
# instantiate a set  -- use s=set()  to instantiate empty set
primes = {2, 3, 5, 7}
evens = {2, 4, 6, 8}

# operators for union, intersection, and difference
even_primes = primes & evens
primes.intersection(evens)

even_or_prime = primes | evens
primes.union(evens)

odd_primes = primes – evens
primes.difference(evens)

not_both = primes ^ evens
primes.symmetric_difference(evens)
```

Dictionary Comprehensions

```
# a very compact way to generate dicts and sets
# syntax is just like list comprehensions except that hash
# tables uses two expressions separated by a colon
hashmap = { key:value for (key,value) in enumerate(iterable) }
myset = { element for key in hashmap.keys() if key%2 == 0 }
```

Strings in Python

```
# can use single or double quotes
s1='foo'
s2="bar"

# access individual characters and subsequences as for lists
print s2[1]                   ==> 'a'
print s1[1:3]                 ==> 'oo'

# strings are immutable, so you need to create a new string:
s2[1] = 'u'                   ==> ERROR
s3 = s2[:1]+'u'+s2[2:]        ==> 'bur'

# concatenation operators
s4 = s1 + s2                  ==> 'foobar'
s5 = 3 * s1                   ==> 'foofoofoo'
```

String Functions in Python

```
s='abccde'; sub='cd'

# string length
print len(s)                    ==> 6

# find substring index
print s.index(sub)              ==> 3

# count occurrences of a substring
print s.count('c')              ==> 2

# substring test
print sub in s                  ==> True

# formatting for output
'Hello {} {}'.format(3,'students')  ==> 'Hello 3 students'
```

see also https://docs.python.org/2/library/stdtypes.html#string-methods

indexing and slicing works exactly as for lists and tuples:

s[1:3] ==> 'bc'
s[-4:] ==> 'ccde'

Python Dictionaries

- **Every** object in Python has an associated hash map which stores the object's *attributes*, called a “dict”
  - attributes include both variables and functions
  - classes are objects with attributes too, so they have a dict as well
- When an object is instantiated, it gets a copy of the class dict
- Two special dicts can be accessed with the **globals()** and **locals()** functions

Python Dicts

```
>>> print globals()
{'__builtins__': <module '__builtin__' (built-in)>,
 '__package__': None, '__name__': '__main__',
 'readline': <module 'readline' from '.../readline.so'>,
 'rlcompleter': <module 'rlcompleter' from '....pyc'>,
 '__doc__': None}

>>> def foo():
...     pass
...
>>> print globals()
{'__builtins__': <module '__builtin__' (built-in)>,
 '__package__': None, '__name__': '__main__',
 'readline': <module 'readline' from '.../readline.so'>,
 'foo': <function foo at 0x7fe0c4846050>,
 'rlcompleter': <module 'rlcompleter' from '....pyc'>,
 '__doc__': None}
```

Python Dicts

```
>>> bar = globals()['foo']
## we've just looked up a function by name!

## and we can invoke the new variable like a function
>>> bar()

# undefine the function:
>>> del globals()['foo']
```

Explicit Type Conversions

```
>>> print 3 + 4
7

# automatic coercion from int to float
>>> print 3 + 4.0
7.0

>>> print 3 + '4'
TypeError: unsupported operand type(s) for +: 'int' and 'str'

>>> print 3 + int('4')
7

Convert to string:     str(int)   or   str(float)
Convert to integer:    int(string)
Convert to float:      float(string)
```

Consequences of “Variables are Pointers”

```
# assigning one variable to another only copies the pointer
x = [1, 2, 3]
y = x
x.append(4)
print y       ==> [1, 2, 3, 4]  # y is changed, too!

# simple types are immutable; arithmetic generates a new object:
x = 10
y = x
x += 3
print y       ==> 10
print x       ==> 13
```

Python Arithmetic Operators

Op	Description
a + b	addition
a - b	subtraction
a * b	multiplication
a / b	division
a // b	integer division (Python3)
a % b	modulus: remainder of a//b
a ** b	exponentiation: **a** raised to **b**
-a	negation
+a	(unary plus) **a** unchanged
a @ b	matrix product (Python 3.5+)

in Python2, / does integer division if both operands are integers

Python Mutation Operators

Op	Description
a += b	add **b** to **a**
a -= b	subtract **b** from **a**
a *= b	multiplication
a /= b	division
a //= b	integer division (Python3)
a %= b	modulus: remainder of a//b
a **= b	raise **a** to the **b** power

- For mutable objects:
  - x = x + y   creates a new object
  - x += y   modifies x "in place"

```
x = [1, 2]
copy = x
x = x + [3]
print x              ==> [1, 2, 3]
print copy           ==> [1, 2]

y = [1, 2]
copy = y
y += [3]
print copy           ==> [1, 2, 3]
```

Python Bitwise Operators

<table>
<tr><th>Op</th><th>Description</th></tr>
<tr><td>a & b</td><td>bitwise AND</td></tr>
<tr><td>a | b</td><td>bitwise OR</td></tr>
<tr><td>a ^ b</td><td>bitwise XOR</td></tr>
<tr><td>~a</td><td>bitwise NOT</td></tr>
<tr><td>a << b</td><td>left shift</td></tr>
<tr><td>a >> b</td><td>arithmetic right shift</td></tr>
</table>

Python Comparison Operators

Op	Description
a == b	equals
a != b	does not equal
a < b	less than
a > b	greater than
a <= b	less than or equal
a >= b	greater than or equal

comparisons can be chained:

15 < a <= 30

Python Boolean Operators

Op	Description
a and b	both **a** and **b** are true
a or b	at least one of **a**, **b** is true
not a	**a** is false

and and **or**
stop evaluating as
soon as the result
is determined

Python Object Operators

Op	Description
a is b	**a** and **b** are the same object
a is not b	**a** and **b** are different objects
a in b	**a** is a member of **b**
a not in b	**a** is not a member of **b**

Flow Control

```
# conditional statements
if COND1:
   block1...
elif COND2:
   block2...
elif COND3:
   block3...
else:
   block4...

# while loops
while COND:
   block...
```

```
# for loops
for VAR in ITERATOR:
   block...

for i in [1,2,3]:
   print i

for i in range(10):
   print(i, end=' ')

for i in range(BEG,END+1,STEP):
   print(i, end=' ')

# within-loop control
break: end loop
continue: skip to next iteration
```

More Flow Control

```
# can detect whether loop exited
#  via break

while i < 10:
   if keyhit():
      break
   else:
      i += 1
else: # no break
   print “No key hit”
```

Defining Functions

```
# use keyword def
# no return type specified
# arguments do not list types
def fib(N):
   L = []
   a, b = 0, 1
   while len(L) < N:
      a, b = b, a + b
      L.append(a)
   return L
fib(5)     ==> [1, 1, 2, 3, 5]

# Python uses “duck typing”:
#  “if it walks like a duck and
#   quacks like a duck, it's a
#   duck”
```

```
# default arg values can be
# specified
def fib(N, a=0, b=1):
   L = []
   while len(L) < N:
      a, b = b, a + b
      L.append(a)
   return L

fib(5)     ==> [1, 1, 2, 3, 5]
fib(5,0,2) ==> [2, 2, 4, 6, 10]
fib(5,3,1) ==> [3, 4, 7, 11, 18]
```

Examples adapted from “A Whirlwind Tour of Python”

Defining Functions

```
# use asterisk for variable num
# of args, double asterisk to
# pass a keyword-value list as
#  a dictionary
def catch_all(*args, **kwargs):
   print(“args = “,args)
   print(“kwargs = “,kwargs)

catch_all(1, 2, 3, a=4, b=5)
==>
args = (1, 2, 3)
kwargs = {'a':4, 'b':5}
```

```
# use lambda function for short,
# inline function definition
add = lambda x,y: x + y

add(1,2)   ==> 3

# we now can pass this function
# to another function!
foo(add,5,6)

# or the inline version:
foo(lambda x,y: x + y,5,6)
```

Examples adapted from “A Whirlwind Tour of Python”

Defining Functions: Caution!

- Function definitions are evaluated when they are encountered in the source file
  - it is possible to redefine functions; the version of function **bar** that function **foo** calls depends on which version of **bar** is current at the time **foo** was called
  - default arguments to a function are evaluated at the time the function definition is evaluated
    - mutable objects can generate unanticipated results

```
# given the definition
def foo(x, lst=[]):
   lst.append(x)
   return lst
```

```
# execute the following seq
print foo(1)     ==> [1]
print foo(2)     ==> [1, 2]
print foo(3)     ==> [1, 2, 3]
print foo(4,[]) ==> [4]
print foo(5,[]) ==> [5]
```

Iterators

```
# Python 2: iterate over a range
# of values
xrange(END+1)
xrange(BEG,END+1,STEP)

# Python 2: generate a list
# Python 3: iterate over a range
# of values
range(END+1)
range(BEG,END+1,STEP)

# return tuples of index,value
# for the elements of the list
enumerate(LIST)
```

```
# apply a function to every
# value of an iterator
square = lambda x: x ** 2
for v in map(square,range(10)):
   print(v, end=' ')

# apply a test to every value of
# an iterator, pass only those
# for which the test is True
even = lambda x: x % 2 == 0
for v in filter(even, range(10)):
   print(v, end=' ')
```

Examples adapted from “A Whirlwind Tour of Python”