Machine Learning in Practice
#1-2: Basic Elements of Python

Sang-Hyun Yoon

Summer 2019

Python Programming Language

- 대중적으로 널리 사용되는 C++와 Java는 문법이 유사
- Python은 C++/Java와 문법이 크게 다름
  - dynamic typing이라 type을 명시하지 않고, LALR(1) parsing을 하지 않아서 괄호대신 indentation으로 hierarchy 표현
- 장점: dynamic typing으로 인해 코드가 짧고 간단하며, 모든 데이터(함수 포함)를 object로 다루어서 프로그래밍 편함
- 단점: dynamic typing으로 인해.. compile-time type checking을 안해서 버그 많음. CPU/memory 사용양이 C++의 100배
  - Python code on 슈퍼컴퓨터 $\approx$ C++ code on 아두이노
- 머신러닝용 언어로 대부분 Python을 쓰는 이유:
  - 개발이 쉬워서 (머신러닝 사용자 중 전산학 전공 비중 적음)
  - (C++로 구현된) TensorFlow와 같은 라이브러리에서 대부분의 자원을 사용하고 CPU보다 GPU 사용량이 더 많아서 Python의 단점이 크게 부각되지 않음
  - 그럼에도 불구하고 TensorFlow는 C++ interface도 제공

Java vs. Python: Poker Example

in Java

```java
// comment by using "//"
class Card {
    int suit, rank;
    public Card (int s, int r) {
        this.suit = s;  this.rank = r; }
}
public class StartUpClass {
    public static bool isFlush (Card[] hand) {
        int[] numCards = new int[4];
        for (int j=0; j<hand.length; j++)
            numCards[hand[j].suit]++;
        for (int i=0; i<4; i++)
            if (numCards[i] >= 5)
                return true;
        return false;
    }
    public static void main (String[] args) {
        Card[] hand = new Card[5];
        for (int i=0; i<5; i++)
            hand[i] = new Card(i%4,i+5);
        System.out.println(isFlush(hand));
    }
}
```

in Python

```python
# comment by using "#"
class Card:
    def __init__ (self, s, r):
        self.suit = s
        self.rank = r


def isFlush (hand):
    numCards = [0] * 4
    for j in range(len(hand)):
        numCards[hand[j].suit] += 1
    for i in range(4):
        if numCards[i] >= 5:
            return True
    return False

def startFromHere():
    hand = []
    for i in range(5):
        hand.append(Card(i%4,i+5))
    print(isFlush(hand))

startFromHere()  # no main method
```

Outline

Dynamic Typing

The biggest difference bet'n Java/Python: static/dynamic typing

- You do not need to specify types (and can't do so)
- Python interpreter will check the type consistency at run-time

Java: Static Typing

```java
boolean function (Card c, int[] a) {
    for (int i=0; i<a.length; i++)
        if (c.rank == a[i])
           return true;
    return false;
}      ...
    int rank = 2;
    int[] a = { 1, 2, 3 };
    Card c = new Card (1, rank);
    if (function (c, a)) ...
```

Python: Dynamic Typing

```python
def function (c, a):
    for i in range(len(a)):
        if c.rank == a[i]:
            return True
    return False
        ...
    rank = 2;
    a = [ 1, 2, 3 ]
    c = Card (1, rank)
    if function (c, a): ...
```

Pros & cons of dynamic typing

- Pros: program becomes simpler, flexible
- Cons: prone to errors (cannot rely on compile-time analysis)

Built-In Primitive Types

Type	Kind of Values	Examples of Value
`int`	integers	`1, 0, -200`
`float`	reals..	`3.14159, 1.234E-8`
`complex`	complex numbers	`3.5+4j`
`str`	strings	`"abc", 'abc'`
`bool`	`True, False`	`x>0 and y<-10`
`NoneType`	`None`	

- No `char` type: ".." and '..' are used interchangeably
- Anyway, you need not explicitly write down the types

- Object types will be introduced later
  - lists/tuples, sets, dictionaries
  - user-defined objects
  - functions (can be used as arguments & return values!)
    - Python provides high-order functions

type keyword

type(..) gives the type of a given variable/value/expression.

```
def function (x):
    if type(x) == int:
        return "integer value"
    elif type(x) == float:
        return "float value"
    elif type(x) == str:
        return "string value"
    elif type(x) == bool:
        return "boolean value"

print(function (1))
print(function (2.5))
print(function ("abc"))
print(function (False))
print(type("3.1415"))
```

Output

```
integer value
float value
string value
boolean value
<type 'str'>
```

Useful when "type-dependent" functions are required.

Type Conversion

```
print(int(17.3))
print(float(17))
print(1 + int("3"))
print(1 + float("3.1415"))
print("a " + str(3.1415))
print(complex(17))
```

Output

```
17
17.0
4
4.1415
a 3.1415
(17+0j)
```

Slightly different from Java, e.g.: `(int)1.5`, `Integer.toString(7)`

Outline

Indentation

- Nested structure is represented by indentation in Python.
  - by braces in Java (indentation for readability only)

Java (good indentation)

```
void function () {
    int sum = 0;
    for (int i=1; i<=9; i++) {
        for (int j=1; j<=9; j++) {
            int v = i*j;
            sum += v;
        }
    }
}
```

Java (bad indent, still works!)

```
void function () {
int sum = 0;
    for (int i=1; i<=9; i++) {
   for (int j=1; j<=9; j++) {
        int v = i*j;
  sum += v;
   }
}
    }
```

Python (correct)

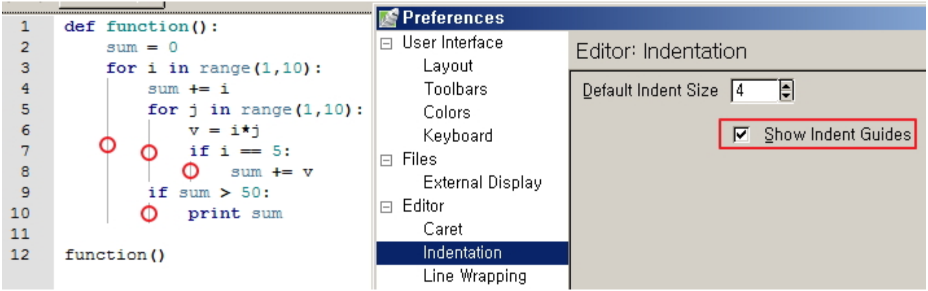
```
def function():
    sum = 0
    for i in range(1,10):
        for j in range(1,10):
            v = i*j
            sum += v
```

Python (does NOT work!)

```
def function():
    sum = 0
    for i in range(1,10):
    for j in range(1,10):
            v = i*j
            sum += v
```

"Indent Guide" functionality in Wing IDE

- Edit ⇒ Preferences ⇒ Editor ⇒ Indentation ⇒ Show Indent Guides
  - recommended indent size: **4**
- Use `Tab`/`Backspace` keys to indent/outdent
  - Do not use space key



Outline

Expressions

Recall: Expression

Legal combination of variables/values & operators

- hour * 60 + minute - 1 + minute / 20
  - illegal expression: hour + + hour minute 60 - / 20

Very similar set of operators as with Java:

- arithmetic: `*`, `/`, `%`, `+`, `-`, `**`, `//`
  - `**`: exponentiation (e.g. `3.5**4.2` ⇒ `192.79..`)
  - 7//3 = 2 (quotient)
- relational: `==`, `!=`, `>`, `<`, `>=`, `<=`
- logical: `and`, `or`, `not`
  - in Java: `&&`, `||`, `!`
- assignment (shortcut): `+=`, `-=`, `*=`, `/=`
  - `++`, `--` not provided

Precedence of Operators

Operators	
`()`	
`**`	↑
`* %`	higher
`+ -`	
`== != > < >= <=`	
`not`	
`and`	lower
`or`	↓
`= += -= *= /=`	

- Tie-breaking rule for the same precedence: from left to right

```
d = x%2 == 1 or x%3 != 0 and (not(y <= 1) or x == 1)
```

- Use ( ) if you want to
  (1) override the precedence rules or
  (2) are not sure what they are!

+, * operators on strings

`str + str` and `str * int` defined in Python:

- `"Hello " + "Python"` ⇒ `"Hello Python"`
- `"Hello" * 4` ⇒ `"HelloHelloHelloHello"`
  - Also, `4 * "Hello"` ⇒ `"HelloHelloHelloHello"`

`str + int` NOT defined in Python

- `"Hello" + 7` not allowed in Python
  - whereas Java allows it

Use type casting `str(.)` to concatenate strings with numbers

- `"Hello" + str(7)` ⇒ `"Hello" + "7"` ⇒ `"Hello7"`

Multiple Assignments (1/2)

Python allows assigning to several variables at the same time:

```
a, b, c = 1, 2, 4
```

Swapping without uncomfortable temporary variables possible!

```
a, b = b, a
x, y, z = y, z, x
p, q = p*q, p-q+1
```

Succinct implementation of gcd using multiple assignments

```
if a < b:
    a, b = b, a
while b != 0:
    a, b = b, a%b
```

Multiple Assignments (2/2)

Multiple assignment across function boundary:

```
def f(x):
  return x+1, x**2, "kamui"


a, b, c = f(10)
print(a)
print(b)
print(c)
```

```
11
100
kamui
```

We'll come back to this issue with tuples (stay tuned)

Outline

Boolean Expressions

Expressions that are evaluated to be `True`/`False` (i.e. `bool` type)

Primitive Boolean Expressions

`x%2 != y+x**2`

- Primitive boolean expressions consists of relational operators:
  - `==`, `!=`, `>`, `<`, `>=`, `<=`

Compound Boolean Expressions

`x == 0 and (y > 10 or x%2 != y+x**2)`

- Primitive expressions can be composed using logical operators:
  - `and`, `or`, `not`

Used in `if`, `for`, `while` statements

if-else statements

```
if boolean_expression :
  STATEMENTS
else:              # else block may be omitted
  STATEMENTS
```

in Java

```
if (x == 1 && y == 1) {
    z = 0;
    z += y;
} else {
    z = 1;
}
if (x+1 != y**2) {
    z += x;
} // no else block
```

in Python

```
if x == 1 and y == 1:
    z = 0
    z += y
else:
    z = 1

if x+1 != y**2:
    z += x
# no else block
```

No () around boolean expr

if-else statements: nested vs. chained (elif keyword)

in Java

```java
// nested conditional
if (x > 0) {
    sign = "positive";
} else {
    if (x < 0) {
        sign = "negative";
    } else {
        sign = "zero";
    }
}

// chained conditional
if (x > 0) {
    sign = "positive";
} else if (x < 0) {
    sign = "negative";
} else {
    sign = "zero";
}
```

in Python

```python
# nested conditional
if x > 0:
    sign = "positive"
else:
    if x < 0:
        sign = "negative"
    else:
        sign = "zero"

# chained conditional
if x > 0:
    sign = "positive"
elif x < 0:
    sign = "negative"
else:
    sign = "zero"
```

Outline

Functions

Differences with Java:

- def keyword declares a function
- Types not specified
- Return values with different types possible!

in Java

```
int func (int a, int b) {
    if (a < 0)
        return "negative"; // error
    if b < 0:
        return;            // error
    return a+b;
}
```

in Python

```
def func (a, b):
    if a < 0:
        return "negative"
    if b < 0:
        return;
    return a+b;

print(func(2,3))    # 5
print(func(-2,3))   # negative
print(func(2,-3))   # None
```

No `main` method, No start-up class

in Java

```
public class StartUpClass {
  public static int func() {
      return 1;
  }

  public static void main(String[] s){
      for (int i=0; i<10; i++)
          func();
  }
}
```

in Python

```
def func():
    return 1

def startFromHere():
    for i in range(10):
        func()

startFromHere()
```

OR

```
def func():
    return 1

for i in range(10):
    func()
```

Recursion

The Ackermann function:

$$A(m,n) = \begin{cases} n+1 & \text{if} \quad m = 0 \\ A(m-1,1) & \text{if} \quad m > 0, n = 0 \\ A(m-1, A(m, n-1)) & \text{if} \quad m > 0, n > 0 \end{cases}$$

in Java

```
int A (int m, int n) {
    if (m == 0)
        return n+1;
    else {
        int x;
        if (n == 0)
            x = A(m-1,1);
        else
            x = A(m-1,A(m,n-1));
        return x;
    }
}
```

in Python

```
def A (m, n):
    if m == 0:
        return n+1
    else:
        if n == 0:
            x = A(m-1,1)
        else:
            x = A(m-1,A(m,n-1))
        return x
```

Useful built-in functions

To use math functions, `import math` modules:

Math functions

```
import math

deg = 45
rad = deg/360.0 * 2 * math.pi
print(math.sin(rad))
print(math.sqrt(math.e))
```

`input()` gets a string on the keyboard (`nextLine()` in Java)

Keyboard input

```
name = input("What is your name?")
print("Hello " + name)

s = input("Enter an integer: ")
n = int(s)  # convert str to int
print(n**2)
```

Default Parameters

Without default parameters

```
def f(x, y):
    print(x, y)

a = f(30, "yellow")
b = f(2,  "yellow")
c = f(28, "silver")
```

With default parameters

```
def f(x = 30, y = "yellow"):
    print(x, y)

a = f()
b = f(2)
c = f(28, "silver")
```

But, `d = f("silver")` not allowed

- Normal/default parameters can be mixed

```
def f(x, y = "yellow"):
    print(x, y)

a = f(30)
b = f(2)
c = f(28, "silver")
```

Outline

while loop

```
while boolean_expression :
  STATEMENTS
```

Counting the number of digits in a positive integer n

in Java

```java
int countDigits (int n) {
    int count = 0;
    while (n > 0) {
        count += 1;
        n = n/10;
    }
    return count;
}
```

in Python

```python
def countDigits (n):
    count = 0
    while n > 0:
        count += 1
        n = n/10

    return count
```

for loop

- for statements significantly different from Java
- Seems less flexible than Java, however, more convenient

```
for item in item_list :
  STATEMENTS (that use item)
```

The simplest for loop pattern

in Java

```
for (int i=0; i<10; i++)
    sum += i;
```

in Python

```
for i in range(10):
    sum += i
```

- range(10) refers to a 'list' [0,1,2,$\cdots$,9]
  - will immediately apppear in the next section
- More patterns will be introduced with sequence/set/dictionary

break / continue

Works exactly the same as in Java

- break terminates the innermost while/for loop
  - that encloses the break.
- continue skips the current iteration of innermost while/for loop
  - that encloses the break.

```
for i in range(10):
    if i == 3:
        print("Skips the iteration" + str(i))
        continue  # skip the current iteration (but not exit loop)

    if a[i] < 0:
        print("Terminates the loop")
        break  # immediately exit the loop

    a[i] -= 1  # skipped in iteration i=3
```

Outline

Handling a collection of data

Java: Arrays

```
int[] a = { 2, 4, 2, 9, 5 };

for (int i=0; i<a.length; i++) {
    sum += a[i];
}
```

Python: Lists

```
a = [ 2, 4, 2, 9, 5 ]

for i in range(len(a)):
    sum += a[i]


print(type(a))    # <type 'list'>
```

- To create a list, enclose the values in `[]`
- `len(·)` denotes the length of a list (c.f. Java: `.length`)
  - c.f. `.length` in Java
- Indexing is the same as in Java
  - Index ranges from `0` (not 1) to `len(·)-1`
- Stronger and more flexible than arrays in Java (stay tuned)

range(·)

Recall the use of the mysterious keyword `range`:

- `range(n)` $\equiv$ `[0,1,2,···,n-1]` (which is a list)
  - In Python 3.x, `list(range(n))` becomes the `[0,1,···,n-1]`

```
for i in range(10):
    sum += i
```

$\equiv$

```
for i in [0,1,2,...,9]:
    sum += i
```

Akin to math expression:

- for each $i \in \{0, 1, \cdots, n-1\}$
  - but explicitly ordered

Now, the meaning of the previous slide's code becomes more clear:

```
a = [ 2, 4, 2, 9, 5 ]

for i in range(len(a)):
    sum += a[i]
```

$\equiv$

```
a = [ 2, 4, 2, 9, 5 ]

for i in [0,1,2,3,4]:
    sum += a[i]
```

Diversion: traversing list in `for` loop (1/2)

Think of the list `S = [ 2, 4, 2, 9, 5 ]` as a set.

1. Traversing list with index (as in the previous slides):

```
for i in range(len(S)):
    sum += S[i]
```

$\equiv$

```
for i in [0,1,2,3,4]:
    sum += S[i]
```

- in math: $\sum_{i \in I} \texttt{S[i]}$ where $I = \{0, 1, \cdots, |\texttt{S}| - 1\}$

2. Traversing list by accessing elements directly:

```
for x in S:
    sum += x
```

$\equiv$

```
for x in [2,4,2,9,5]:
    sum += x
```

- in math: $\sum_{x \in S} \texttt{x}$
  - conceptually more clear/succinct!

When is the former more appropriate? When the latter?

Diversion: traversing list in `for` loop (2/2)

Sum of list elements: Traversing order not important

Using index

```
for i in range(len(S)):
    sum += S[i]
```

vs.

Direct access

```
for x in S:
    sum += x
```

Testing if monotonically increasing: Traversing order matters!

Using index

```
for i in range(len(S)-1):
    if S[i] > S[i+1]:
        return False
return True
```

vs.

Direct access

```
for x in S:
    if x > ??? :
        return False
return True
```

- If index seems inevitably necessary, use index-based style.
- Otherwise, use direct-access style.

Lists can contain elements of any types!

- List of strings:

```
[ "Kakashi", "Jiraiya", "Guu", "Pokute" ]
```

- List of elements with different types (c.f. Java):

```
[ 1, True, 3.5, "Madara"]
```

- List of lists (recursively):
  - A multidimensional arrays can be modeled by a list of lists

```
[ [1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12] ]  # 4x3
[ [1, 2], ["abc", 3.5], [ [4, 5], [7, [8, 9], 10] ] ]
```

- List of functions:

```
[ add, subtract, isFullHouse, countPrime ]
```

Much more flexible and expressive than arrays in Java!

range() : revisited

- `range(n)` $\equiv$ `[0,1,2,···,n-1]`
- `range(m,n)` $\equiv$ `[m,m+1,···,n-1]`
  - `range(m,n+1)` $\equiv$ `[m,m+1,···,n]`
- `range(m,n,k)` $\equiv$ `[m,m+k,m+2k,···]`
  - `range(3,20,5)` $\equiv$ `[3,8,13,18]`
  - `range(20,-7,-5)` $\equiv$ `[20,15,10,5,0,-5]`

in Java

```
for (int i=10; i<=20; i+=3)
    sum += i

for (int i=10; i>=0; i-=1)
    sum += i
```

in Python

```
for i in range(10,21,3):
    sum += i

for i in range(10,-1,-1):
    sum += i
```

c.f. `[1,4,9,16,···,100]`?

- `[i*i for i in range(11)]` (will be introduced later)

Creating uninitialized lists

Java: Array with size specified

```
int[] a = new int[n];
for (int i=0; i<a.length; i++) {
    a[i] =  i*i+1;
}
```

Python: List with size specified

```
a = [0] * n     # or [None] * n
for i in range(len(a)):
    a[i] = i*i+1;
```

OR

Python: Variable-sized list

```
a = []
for i in range(len(a)):
    a.append(i*i+1);
```

- `[x] * n` ≡ `[x,x,···,x]` (n times)
- `[]` : empty list (math analogy: empty set ϕ)
  - `.append()` add an element at the end of the list
- `[[x] * n for i in range(m)]` : 2-dimensional m×n lists
  - `[[[x] * n for i in range(m)] for j in range(k)]` : k×m×n

Aliasing vs. Copy

Exactly the same as in Java!

Java

```
int[] a = { 0, 1, 2, 3, 4 };

// aliasing
int[] b = a;
b[0] = 10;   // a[0] = 10
System.out.println(a[0]);

// copy
int[] c = new int[a.length];
for (int i=0; i<a.length; i++)
    c[i] = a[i];
c[1] = 20;
System.out.println(a[1]);
```

```
10
1
```

Python

```
a = [ 0, 1, 2, 3, 4 ]

# aliasing
b = a
b[0] = 10   // a[0] = 10
print(a[0])

# copy
c = [None] * len(a)
for i in range(len(a)):
    c[i] = a[i]
c[1] = 20
print(a[1])
```

```
10
1
```

Aliasing vs. Copy: Shallow Equality vs. Deep Equality

- Shallow equality operator: is in Python, == in Java
- Deep equality operator: == in Python, equals in Java

```
// aliasing
int[] b = a;
..print(b == a);     // shallow
..print(equals(b, a)); // deep

// copy
int[] c = new int[a.length];
for (int i=0; i<a.length; i++)
    c[i] = a[i];
..print(c == a);     // shallow
..print(equals(c, a)); // deep
```

```
true
true
false
true
```

```
# aliasing
b = a
print(b is a)   # shallow
print(b == a)   # deep

# copy
c = [None] * len(a)
for i in range(len(a)):
    c[i] = a[i]
print(c is a)   # shallow
print(c == a)   # deep
```

```
True
True
False
True
```

Lists as Parameters & Return Values

Java

```
int[] copy (int[] b) {  // b = a
   // b is aliased to a

   int[] c = new int[b.length];
   for (int i=0; i<c.length; i++)
       c[i] = b[i];

   b[0] = 10;  // a[0]=10 (alias)

   return c;
}
      ...
int[] a = { 0, 1, 2, 3, 4 };
int[] d = copy (a);
..println(a[0]);  // 10
..println(d[0]);  // 0
```

Python

```
def copy (b):  # b = a
   # b is aliased to a

   c = [None] * len(b)
   for i in range(len(c)):
       c[i] = b[i]

   b[0] = 10;  # a[0]=10 (alias)

   return c


a = [ 0, 1, 2, 3, 4 ]
d = copy (a);
print(a[0])    # 10
print(d[0])    # 0
```

Call-by-reference for lists (as with arrays in Java)

Call-by-Value vs. Call-by-Reference

The same as in Java:

- Call-by-value: primitive types
- Call-by-reference: object types (including lists)

Call-by-Value

```
def increment (x):
    x = x+1

a = 10
increment (a)
print(a)          # output: 10 (but not 11)
```

Call-by-Reference

```
def increment (x):
    x[0] = x[0]+1

a = [10, 20, 30]
increment (a)
print(a[0])         # 11
```

Built-in operators/functions for lists: non-object style

- + operator: concatenates lists

```
[1,2,"abc"] + [] + [5,[6,7]]    # [1,2,"abc",5,[6,7]]
```

- `len(·)`: the length of a list

```
a = [1, True, 3.5, "Itachi"]
len(a)   # 4
```

Functions for lists of numbers (`int`/`float`):

- `sum(·)`: the sum of elements of a list
- `max(·)`: the largest element
- `min(·)`: the smallest element

```
a = [3, 4.5, 6.7]
sum(a)   # 14.2
max(a)   # 6.7
min(a)   # 3
```

Slicing & Copy

Slicing `b = a[i:j]` creates a new sub-list b of the list a:

- b contains elements i, i+1, ⋯, j-1 of a
- If i is omitted, the sub-list starts with the first element
- If j is omitted, the sub-list end with the last element

We can create a copy (not alias) of by

- b = a[:] (c.f. b = a creates an alias only)
  - but not completely deep copy for multi-dimensional lists

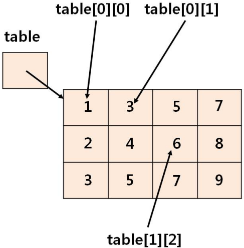
```
a = [0, 10, 20, 30, 40, 50, 60]

b = a[2:5] ; print(b)
b = a[:5]  ; print(b)
b = a[2:]  ; print(b)

b = a[:]  ;  print(b)
print(b == a)
print(b is a)
```

```
[20, 30, 40]
[0, 10, 20, 30, 40]
[20, 30, 40, 50, 60]

[0, 10, 20, 30, 40, 50, 60]
True
False
```

2-dimensional array as a list

- Useful in representing matrices
- `table[i]` is the `row-i` list
- `len(table)` is height and `len(table[0])` is width

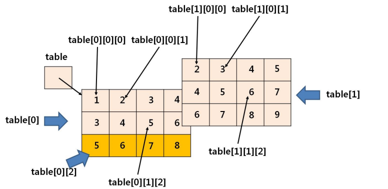
```
height = 3    # = number of rows = size of a column
width = 4     # = number of columns = size of a row
table = [ [None] * width  for i in range(height)]

for i in range(height):
    for j in range(width):
        table[i][j] = (i+2*j+1)
```



3-dimensional array as a list

- table[i] is the floot-i 2D array
- table[i][j] is the floor-i/row-j 1D array
- len(table) is depth and len(table[0]) is height ...

```
height = 3    # = number of rows
width = 4     # = number of columns
depth = 2     # = number of floors
table = [ [ [None] * width  for j in range(height)]
                                for i in range(depth) ]
for i in range(depth):
    for j in range(height):
        for k in range(width):
            table[i][j][k] = (i+2*j+1+k)
```



Outline

Tuples

- Tuples can be thought of as immutable lists
- A tuple shares all the characteristics of a list except that violates immutability

List

```
a = [ 2, 4, 2, 9, 5 ]
print(type(a)) # <class 'list'>

sum = 0
for i in range(len(a)):
    sum += a[i]

for x in a:
    sum += x

a[1] = 10    # legal
```

Tuple

```
a = ( 2, 4, 2, 9, 5 )
print(type(a)) # <class 'tuple'>

sum = 0
for i in range(len(a)):
    sum += a[i]

for x in a:
    sum += x

a[1] = 10    #  error incurred
```

Creating tuples

- Can omit parenthesis ()
  - Packing done automatically
- Single-element tuple must be created with `(x,)`
  - `(x)` simply yields `x`

```
a = 2, 4, 2, 9, 5
print(a)

b = (1)  ;  print(b)
c = (1,) ;  print(c)
d = 1,   ;  print(d)
```

```
(2, 4, 2, 9, 5)  # automatic packing

1       # not a tuple
(1,)    # comma makes it a tuple
(1,)    # still can omit ()
```

- As with the list, a tuple can contain elements of any type

```
( (1, 2), ["abc", 3.5], [ (4, 5), [7, (8, 9), 10] ] )
```

Tuple assignment

- `(a,b,c) = (1,2,3)` is allowed in Python

Tuple Assignment

```
(a, b, c) = (1, 2, 4)
(b, c, a) = (c, a, b)

if a < b:
    (a, b) = (b, a)
while b != 0:
    (a, b) = (b, a%b)
```

Recall: Multiple Assignment

```
a, b, c = 1, 2, 4
b, c, a = c, a, b

if a < b:
    a, b = b, a
while b != 0:
    a, b = b, a%b
```

- Essentially, multiple assignment $\equiv$ tuple assignment
  - `a,b,c` is automatically packed into the tuple `(a,b,c)`
- The number of elements in LHS/RHS tuples must be equal
  - `(a,b) = (1,2,3)` is illegal
- Every element of LHS tuple must be variables
  - `(a,b+1) = (1,2)` is illegal

Tuple assignment

The following are the same (automatically converted into tuples):

- `a,b,c = x,y,z`
- `a,b,c = (x,y,z)`
- `(a,b,c) = x,y,z`
- `(a,b,c) = (x,y,z)`
- `t = (x,y,z)`
  `a,b,c = t`
- `t = x,y,z`
  `(a,b,c) = t`
- `t = x,y,z`
  `a,b,c = t`
- `t = (x,y,z)`
  `(a,b,c) = t`

Tuple assignment across function boundary:

```
def f(x):
  return x+1, x**2, "kamui"

a, b, c = f(10)
print(a)
print(b)
print(c)
```

```
11
100
kamui
```

Operations for tuples

Operations for lists also work for tuples, except those that violate immutability:

- `+` (concatenate), `*` (repeat)
- `a[i:j]` (slicing), indexing for reading (but not for writing)
- `in` (membership), `for` loop
- `==` (deep equality), `is` (shallow equality)
- `len()`, `min()`, `max()`, `sum()`

```
a = (0,1,2,3)
print(a+a) # (0,1,2,3,0,1,2,3)
print(a*2) # (0,1,2,3,0,1,2,3)
print(a[1])     # 1
print(a[:2])    # (0,1)
print(a[1:2])   # (1,)
print(2 in a)   # True
print(7 in a)   # False
```

```
for x in a:
    print(x, end=" ")
# 0 1 2 3

print(len(a))  # 4
print(min(a))  # 0
print(max(a))  # 3
print(sum(a))  # 6
```

Operations that do not work for tuples

None of the operations that change lists are available for tuples:

- `a[i] = x` (indexing for writing)
- `a.append()`
- `a.reverse()`
- `a.sort()`
- `a.extend()`
- `a.insert()`
- `a.remove()`
- `a.pop()`

Tuples as elements of `for` loops

```
a = [(1,2,"abc"), (3,4,(5,6)), (7,True,[8,9])]
for (x,y,z) in a:
    print(z)
```

- tuple assignment `(x,y,z) = (1,2,"abc")` invoked

```
a = [(1,2,"abc"), (3,4,(5,6)), (7,True,[8,9])]
for x,y,z in a:
    print(z)
```

- `x,y,z = (1,2,"abc")` invoked
  - still work since `x,y,z` is converted into `(x,y,z)`

Commas in `print` function

```
print( (1, "abc", (2, 3)) )
print( 1, "abc", (2, 3) )
```

```
(1, 'abc', (2, 3))
1 abc (2, 3)
```

- Commas in `print` statement do not convert expressions into a tuple
- Have the effect of printing expressions in horizontal line

Conversion between lists/tuples

Lists and tuples can be converted to each other

```
s1 = [2, 3, 5]         # list
print(tuple(s1))

s2 = (2, 3, 5)         # tuple
print(list(s1))
```

```
(2, 3, 5)


[2, 3, 5]
```

When these conversions useful?

```
a = (6, 1, 4, 3)
# a.sort() incurrs error (since tuple is immutable)

b = list(a)
b.sort()

a = tuple(b)
```

Why tuples?

Lists alone are sufficient to implement every functionality.

Why tuples as a separate type for "immutable lists"?

- For integrity & persistence, i.e. to prevent a tuple from being changed.
- Can be used in a few places where mutable objects are not allowed, e.g. sets/dictionaries (stay tuned)

Outline

User-Defined Objects

in Java

```java
class Card {
    int suit, rank;
    public Card (int s, int r) {
        this.suit = s;  this.rank = r; }
}

public class StartUpClass {
    ..void funcA (Card[] hand, Card c) {
        for (int i=0; i<hand.length; i++)
            hand[i].suit = c.suit;
    }
    ..void funcB () {
        Card[] hand = new Card[5];
        for (int i=0; i<5; i++)
            hand[i] = new Card(i%4,i+5);
        Card c = new Card(0,3);
        funcA (hand, c);
    }
}
```

in Python

```python
class Card:
    def __init__ (self, s, r):
        self.suit = s
        self.rank = r



def funcA (hand, c):
    for card in hand:
        card.suit = c.suit

def funcB ():
    hand = [None] * 5
    for i in range(5):
        hand[i] = Card(i%4,i+5))
    c = Card(0,3)
    funcA (hand, c)
```

Defining an object type & using constructors to create objects

in Java

```
class Card {
    int suit, rank;
    public Card () { suit = rank = 0; }
    public Card (int s, int r) {
        this.suit = s;  this.rank = r; }
}
    ..void funcB () {
        Card[] hand = new Card[5];
        for (int i=0; i<5; i++)
            hand[i] = new Card(i%4,i+5);
        Card c = new Card(0,3);
```

in Python

```
class Card:
    def __init__ (self, s, r):
        self.suit = s
        self.rank = r


def funcB ():
    hand = [None] * 5
    for i in range(5):
        hand[i] = Card(i%4,i+5))
    c = Card(0,3)
```

- Unique constructor `__init__` (multiple `__init__` not allowed)
  - Exploit default parameters in `__init__`
- The first parameter of every constructor must be `self`
  - `self` has the same role as `this` in Java
- Instance variables not declared in class body, only in `__init__`
- `new` is omitted in Python

Objects as Parameters & Return Values

in Java

```java
..Card funcA (Card c) {
    c.suit = 2;  // call-by-reference
    Card d = new Card (0, c.rank);
    return d;
}
..void funcB () {
    Card c = new Card(0,3);
    Card e = funcA (c);
    System.out.println(c.suit); // 2
```

in Python

```python
def funcA (c):
    c.suit = 2  # call-by-refere
    d = Card (0, c.rank)
    return d

def funcB ():
    c = Card(0,3)
    e = funcA (c)
    print(c.suit)  # 2
```

- Can pass objects as parameters or return values
- Call-by-reference for objects (as in Java)
    - Lists in Python (& arrays in Java) are also called-by-reference

Lists of Objects

in Java

```
..funcA (Card[] hand, Card c) {
    for (int i=0; i<hand.length; i++)
        hand[i].suit = c.suit;
}
..void funcB () {
    Card[] hand = new Card[5];
    for (int i=0; i<5; i++)
        hand[i] = new Card(i%4,i+5);
    Card c = new Card(0,3);
    Card d = funcA (hand, c);
```

in Python

```
def funcA (hand, c):
    for card in hand:
        card.suit = c.suit

def funcB ():
    hand = [None] * 5
    for i in range(5):
        hand[i] = Card(i%4,i+5))
    c = Card(0,3)
    d = funcA (hand, c)
```

- Very similar to arrays of objects in Java
- Objects of lists also possible (as objects of arrays in Java)

You can define semantics of built-in operations as you wish!

```
class Rational(object):
  def __init__(self, numer, denom):
    self.n = numer
    self.d = denom

  def __add__(self, r):
    n = self.n * r.d + r.n * self.d
    d = self.d * r.d
    return Rational(n,d)

  def __mul__(self, r):
    n = self.n * r.n
    d = self.d * r.d
    return Rational(n, d)

  def __abs__(self):
    n = abs(self.n)
    d = abs(self.d)
    return Rational(n, d)
```

```
r1 = Rational(1,2)
r2 = Rational(2,5)

r3 = r1 + r2
# r3 = r1.__add__(r2)

r4 = r3 * r1
# r4 = r3.__mul__(r1)

r5 = abs(r4)
# r5 = r4.__abs__()
```

Operator Overloading: What For?

With operator overloading

```
class Rational(object):
  def __add__(self, r):
    ...
  def __mul__(self, r):
    ...
  def __abs__(self):
    ...
  def __neg__(self):
    ...
  def __pow__(self, p):
    ...

r5 = -r1/r3 + r2*r4
r6 = r4*(r1**2-abs(r3))
```

Without operator overloading

```
class Rational(object):
  def add(self, r):
    ...
  def mul(self, r):
    ...
  def abs(self):
    ...
  def neg(self):
    ...
  def pow(self, p):
    ...

r5 = r1.neg().div(r3).add(r2.pow(4))
r6 = r4.mul(r1.pow(2).sub(r3.abs()))
```

- Which one seems better and more convenient to use?
- Operator overloading allows us to use infix notations for built-in operations, which is much more intuitive

Overloadable Built-In Operations

Operation	As Function	Description
x + y	x.__add__(y)	addition
x - y	x.__sub__(y)	subtraction
x * y	x.__mul__(y)	multiplication
x / y	x.__truediv__(y)	division
- x	x.__neg__()	unary minus
abs(x)	x.__abs__()	absolute value
x ** n	x.__pow__(n)	exponent
x == y	x.__eq__(y)	equality
x != y	x.__ne__(y)	not equal
x > y	x.__gt__(y)	greater than
x >= y	x.__ge__(y)	greater than or equal
x < y	x.__lt__(y)	less than
x <= y	x.__le__(y)	less than or equal
len(x)	x.__len__()	length of the sequence
x in y	x.__contains__()	does the sequence y contain x?
x[key]	x.__getitem__(key)	access element key of sequence x
x[key] = y	x.__setitem__(key, y)	set element key of x to value y
str(x)	x.__str__() / x.__repr__()	convert to a printable string

Does == check deep equality for user-defined objects?

Recall: == works as deep equality for

- lists (of lists (of lists ...)), tuples, strings, sets, dictionaries

However, == does NOT works only as deep equality for

- user-defined objects (only works as shallow equality)

```
print(Rational(1,2) == Rational(1,2))   # False
```

Fortunately, == can be redefined by __eq__ as we wish!

```
class Rational:
    def __eq__(self, r):
        return self.n * r.d == r.n * self.d

print(Rational(1,2) == Rational(1,2))   # True
print(Rational(1,2) == Rational(3,6))   # True
```

Outline

Comments

Java

```java
// blah blah
System.out.println("Kakashi"); // blah
  /* blah blah blah blah blah blah blah blah blah blah
     blah blah blah blah blah blah blah blah blah blah
     blah blah */
System.out.println(/* blah */ "Kamui");
```

Python

```python
# blah blah
print("Kakashi") # blah
  """ blah blah blah blah blah blah blah blah blah blah
     blah blah blah blah blah blah blah blah blah blah
     blah blah """
print(""" blah """ "Kamui")
```

Use `"""..."""` for block comment

Line Continuation

Unlike in Java, Python requires specifying continuation mark \

```
if a == True and \
   b == False:
```

```
y = f(x1, x2, x3, \
      x4, x6, x6)
```

```
y = 1 + 2 + 3 + \
    4 + 5 + 6
```

print without line break

print with comma prints in horozontal line

```
for i in range(3):
    print(i, end=" ")

for i in range(3):
    print(i)
```

```
0 1 2



0
1
2
```

```
x = 10
print(x, "+ 1 =", x+1
```

```
10 + 1 = 11
```

Single-line compound statements

If function, `for`, `if-else` consists of only one statement, it can be written in one line

```
def f(x): return x+1

if x==0:  x += 1
elif x==1:  x += 2
else: x += 3

for i in range(10):  sum += i
```

```
def f(x):
    x += 1
    return x
```

```
def f(x): x += 1
    return x
# error!
```

Empty statement with pass

pass is useful when developing programs incrementally

Java

```
void f(int x) {
}
```

Python (incorrect)

```
def f(x):
            # error!
```

Python (correct)

```
def f(x):
    pass
```

```
if (x == 0)
    ;
else
    x += 1;
```

```
if x == 0:
            # error!
else
    x += 1:
```

```
if x == 0:
    pass
else
    x += 1:
```

Outline

Integer Division (Quotient)

- In Python 3, / is not an integer division (quotient) operator
  - more convenient for numerical applications

Python 2

```
5/2     # 2
float(x)/y

5//2    # 2
5.0/2   # 2.5

5.0//2.4  # 2.0
```

Python 3

```
5/2     # 2.5          (float div)
x/y     # float(.) not needed

5//2    # 2             (int div)
5.0/2   # 2.5

5.0//2.4  # 2.0       (quotient)
```

Console Input/Output

- In Python 3, `print` is a function (not a command)

Python 2

```
s = raw_input("Enter name")
n = raw_input("Enter age")

print s, n
print s,    # line unchanged
print
```

Python 3

```
s = input("Enter name")
n = input("Enter age")

print(s, n)
print(s, end=" ")
print()
```

Lists vs. Views/Iterators : range

- In Python 3, range(·) is not a list
  - Use type conversion with list(·)

Python 2

```
L = range(10)   # list
L.reverse()
```

Python 3

```
L = range(10)   # L is not a list
L.reverse()     # error

L = list(range(10))
L.reverse()
```

- No xrange(·) in Python 3
- range(·) in Python 3 $\equiv$ xrange(·) in Python 2

Python 2

```
for i in xrange(10):
    ...
```

Python 3

```
for i in xrange(10): # error
    ...
```

Lists vs. Views/Iterators : Dictionaries

- In Python 3, `.keys()`, `.values()` `.items()` are not lists
  - Use type conversion with `list(·)`

Python 2

```
d = dict() ...
for key in d.keys():
    ...

L = d.keys()  # list
L.sort()
```

Python 3

```
d = dict() ...
for key in d.keys():
    ...

L = d.keys()    # L is not a list
L.sort()        # error

L = list(d.keys())
L.sort()
```

Lists vs. Views/Iterators : map/filter

- In Python 3, return type of map(·)/filter(·) is not list
  - Use type conversion with list(·)

Python 2

```
M = [3,1,5,4,2]
f = lambda x: x*2

L = map(f, M)  # list
L.sort()
```

Python 3

```
M = [3,1,5,4,2]
f = lambda x: x*2

L = map(f, M)  # L is not a list
L.sort()       # error

L = list(L)
L.sort()
```

Tuple Parameter Unpacking

- In Python 3, tuple parameter unpacking not supported

Python 2

```
def f((r,g,b), (x,y)):
    ...




c = (152,15,102)
p = (24,46)
f(c, p)
```

Python 3

```
def f(color, pos):
    r,g,b = color
    x,y = pos
    ...


c = (152,15,102)
p = (24,46)
f(c, p)
```

Sorting under User-Defined Total Order Relations

User-defined total order

```
def myCmp(x,y): return -1, 1, or 0 ...
```

Python 2

```
L.sort(myCmp)
```

Python 3

```
from functools import cmp_to_key
L.sort(key=cmp_to_key(myCmp))
```

- In Python 3, only key functions are accepted by `sort/sorted`
- The built-in `cmp` function is removed
- `cmp_to_key` converts a comparison function to a key function

Python 2 & 3

```
L = [[1,1], [-1,-1], [-1,0], [0,-1]]
f = lambda p: math.atan2(p[1],p[0])
L.sort(key=f)
```

Operator Overloading: Division & Comparison

- In Python 3, `__cmp__` is no longer supported (Oops!)

Python 2

```
class Rational(object):
  def __div_(self, r): ...

  def __cmp__(self, r):
    # subsumes ==/!=/<=/>=/</>
    # return 0 if self == r
    # return > 0 if self > r
    # return < 0 if self < r
```

Python 3

```
class Rational(object):
  def __truediv__(self, r): ...

  def __eq__(self, r): ...   # ==
  def __ne__(self, r): ...   # !=
  def __le__(self, r): ...   # <=
  def __ge__(self, r): ...   # >=
  def __lt__(self, r): ...   # <
  def __gt__(self, r): ...   # >
```

- For sorting, only `__lt__` needed, and for hashing, only `__eq__`

```
class Point(object):
  def __lt_(self, p):  # no other ops needed for sorting
    return self.x < p.x or (self.x == p.x & self.y > p.y)
...
points.sort()
```

Misc.

- In Python 3, reduce(·) is moved to the functools module

Python 2

```
f = lambda x,y: x+y
reduce(f, [3,1,4,2])
```

Python 3

```
f = lambda x,y: x+y
import functools
functools.reduce(f, [3,1,4,2])
```

- Default printing format for set is {· · ·} instead of set([· · ·])

```
print {1,2,3}
>> set([1,2,3])
```

```
print(set([1,2,3]))
>> {1,2,3}
```

- sys.maxint is replaced by sys.maxsize

```
minVal = sys.maxint
for x in L:
  minVal = min(minVal, x)
```

```
minVal = sys.maxsize
for x in L:
  minVal = min(minVal, x)
```